1984

Design and analysis of ordering and join operations for a multi-backend database system.

Muldur, Serdar

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND ANALYSIS OF ORDERING AND JOIN FOR A MULTI-BACKEND DATABASE SYSTEM

by

Serdar Muldur

June 1984

Thesis Advisor: David K. Hsiao

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design and Analysis of Ordering and Join Operations for a Multi-Backend Database System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1984 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Serdar Muldur		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE June 1984 13. NUMBER OF PAGES 106
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

effective complexity, computing complexity, access complexity, communication complexity

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis proposes implementations of the sort and join in the Multi-Backend Database System.. The idea of implementing these operations is to provide better support for relational databases and relational language interfaces. The key issue analyzed is the distribution of functionality of the operation across the multiple minicomputers of the MDBS architecture. The join analysis also examines alternative join algorithms.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Design and Analysis of Ordering and Join Operations for a Multi-Backend Database System

by

Serdar Muldur
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

This thesis proposes implementations of the sort and join in the Multiple-Backend Database System. The idea of implementing these operations is to provide better support for relational databases and relational language interfaces. The key issue analyzed is the distribution of functionality of the operation across the multiple minicomputers of the MDBS architecture. The join analysis also examines alternative join algorithms.

TABLE OF CONTENTS

I. INTRODUCTION 8

A. THE ORGANIZATION OF THE THESIS 9

II. A REVIEW OF THE MDBS HARDWARE AND SOFTWARE ARCHITECTURES 11

A. DESIGN GOALS FOR MDBS..... 11

B. THE MDBS SOFTWARE ARCHITECTURE 14

III. ALTERNATIVES FOR DISTRIBUTING THE SORTING FUNCTION 13

A. ASSUMPTIONS 19

B. NOTATION 20

C. SYNTAX FOR THE SORT FUNCTION 21

IV. THE ISSUE OF DISTRIBUTION OF FUNCTIONALITY 24

A. THE CONTROLLER PERFORMS THE SORT FUNCTION .. 25

B. THE BACKENDS PERFORM THE SORT FUNCTION 27

1. All Backends Sort, and One or Two Backends Merge 27

2. All Backends Sort Separately and Share Merging 29

C. THE CONTROLLER AND THE BACKENDS SHARE THE SORT FUNCTION 33

1. Backends Sort Block-by-Block and Controller Merges 33

2. The Backends Sort and Perform a Partial Merge, and the Controller Performs the Final Merge 34
D. EVALUATING THE ALTERNATIVES 36
E. COMPARISONS BETWEEN ALTERNATIVES C.1 AND C.2 41
F. RECOMMENDED DISTRIBUTION OF FUNCTIONALITY .. 45
V. DIFFERENT ALGORITHMS FOR THE SORT AND MERGE PHASES 48
A. SORTING WITH n BLOCKS AT A TIME 48
B. THE K-WAY MERGE 52
C. FITTING THE SOFTWARE ARCHITECTURE OF MDBS .. 53
1. Utilizing the Descriptor and Cluster Information 58
2. Utilizing Existing Mechanism for Storing Temporary Data 60
3. The Case that Records are not Evenly Distributed Across the Backends 62
a. The Backend Performs the Sort Function 63
b. The Records are Distributed Evenly Among the Other Backends .. 64
VI. INTRODUCTION TO THE JOIN 65
A. TERMINOLOGY AND NOTATION 65
B. ASSUMPTIONS 69
C. A SYNTAX FOR THE JOIN 70

VII. THE ALTERNATIVE DISTRIBUTIONS OF THE JOIN FUNCTION 72

A. THE CONTROLLER PERFORMS THE JOIN FUNCTION 72

B. THE BACKENDS PERFORM THE JOIN FUNCTION 73

1. The Backends Share the Join Equally 74

2. The Backends Perform the Join Step-by-Step 75

3. One Backend Performs the Join Function 79

C. THE CONTROLLER AND THE BACKENDS SHARE THE JOIN FUNCTION 80

D. EVALUATING THE ALTERNATIVE DISTRIBUTIONS OF THE FUNCTIONALITY 82

VIII. AN ALTERNATIVE JOIN ALGORITHM 85

A. ALTERNATIVE DISTRIBUTIONS OF THE JOIN FUNCTION BY USING A SORT-MATCH ALGORITHM ... 86

1. The Backends Share the Join 87

2. The Controller Performs the Join 90

B. COMPARISONS BETWEEN THE TWO ALTERNATIVES ... 92

C. COMPARISONS BETWEEN THE STRAIGHTFORWARD AND THE SORT-MATCH JOIN ALGORITHMS 96

D. RECOMMENDED PROPOSAL FOR THE DISTRIBUTION OF THE JOIN OPERATION 98

IX. CONCLUSION 103

LIST OF REFERENCES 105

INITIAL DISTRIBUTION LIST 106

I. INTRODUCTION

A current research effort at the Naval Postgraduate School is the investigation of the idea of a database kernel. It is proposed that the attribute-based data model and the attribute-based data language (ABDL) is used as a kernel to support relational, hierarchical, and network databases. A prototype software database system, the Multi-Backend Database System (MDBS), which uses the attribute-based data model, is the target kernel system.

The operations of the attribute-based data language are RETRIEVE, INSERT, DELETE, and UPDATE, the four primary operations of any database management. One proposal is that additional operations be implemented in MDBS to provide a more complete database kernel. In this thesis, we investigate the addition of a sorting capability and the relational join operation.

MDBS is a multiple-processor system. The interesting issue, when considering the implementation of the sort and join operations, is the distribution of functionality among the multiple processors. In this thesis, we propose and analyze various distributions of the functionality.

In analyzing the issues of alternative distributions of the functions, our approach will be to use the existing functional units in MDBS. We propose alternatives, and evaluate them according to the design goals of MDBS. Our proposals require minimal interface changes among the functional units.

We will approach the issues in the following manner. We will make a number of proposals. We will analyze the time complexity of the proposals. Then, based on the MDBS design goals and the complexity analyses, we will make specific recommendations.

A. THE ORGANIZATION OF THE THESIS

In the rest of the thesis, we examine the distribution of functionality for the sort and join operations. Specifically, Chapters II through V cover the sort function. Then, Chapters VI through VIII cover the join.

In Chapter II, we give a brief review of the MDBS hardware and sotware architectures. In Chapter III, we present the general assumptions and notation used in analyzing the alternatives. In Chapter IV, we consider the distribution of functionality among the controller and the backends. In Chapter V, we consider specific algorithms for introducing the sorting function. We also examine the case where a particular sorting task does not fit the MDBS architecture. We discuss how the sort function might

incorporate features of the MDBS software architecture as well.

Chapter VI introduces the join. In Chapter VII, we examine the alternative distributions of the join function among the controller and the backends. In Chapter VIII, a specific join algorithm, the sort-match join algorithm, is examined in the context of MDBS. Finally, in Chapter IX, we summarize our conclusions and discuss the contributions of the thesis.

II. A REVIEW OF THE MDBS HARDWARE AND SOFTWARE ARCHITECTURES

MDBS is a multiple minicomputer system that uses off-the-shelf hardware and special-purpose software in an innovative configuration to support high-performance database operations and large-capacity databases. An overview of the MDBS hardware organization is shown in Figure 2.1. The backends and the controller, which are general-purpose minicomputers, are connected by a broadcast bus. The controller will broadcast each request to all backends at the same time. The backends process the request, and send the results to the controller via the broadcast bus. Intercommunication between the backends is also via the broadcast bus. Every backend has its own dedicated disk drives. Reader should refer to [Ref. 1, 2, 3] for more detail.

A. DESIGN GOALS FOR MDBS

The major problem for conventional database systems is their inability to achieve high performance as the database grows and the rate of requests increases. In order to overcome this problem, a high-performance multi-backend database system should have the following properties.

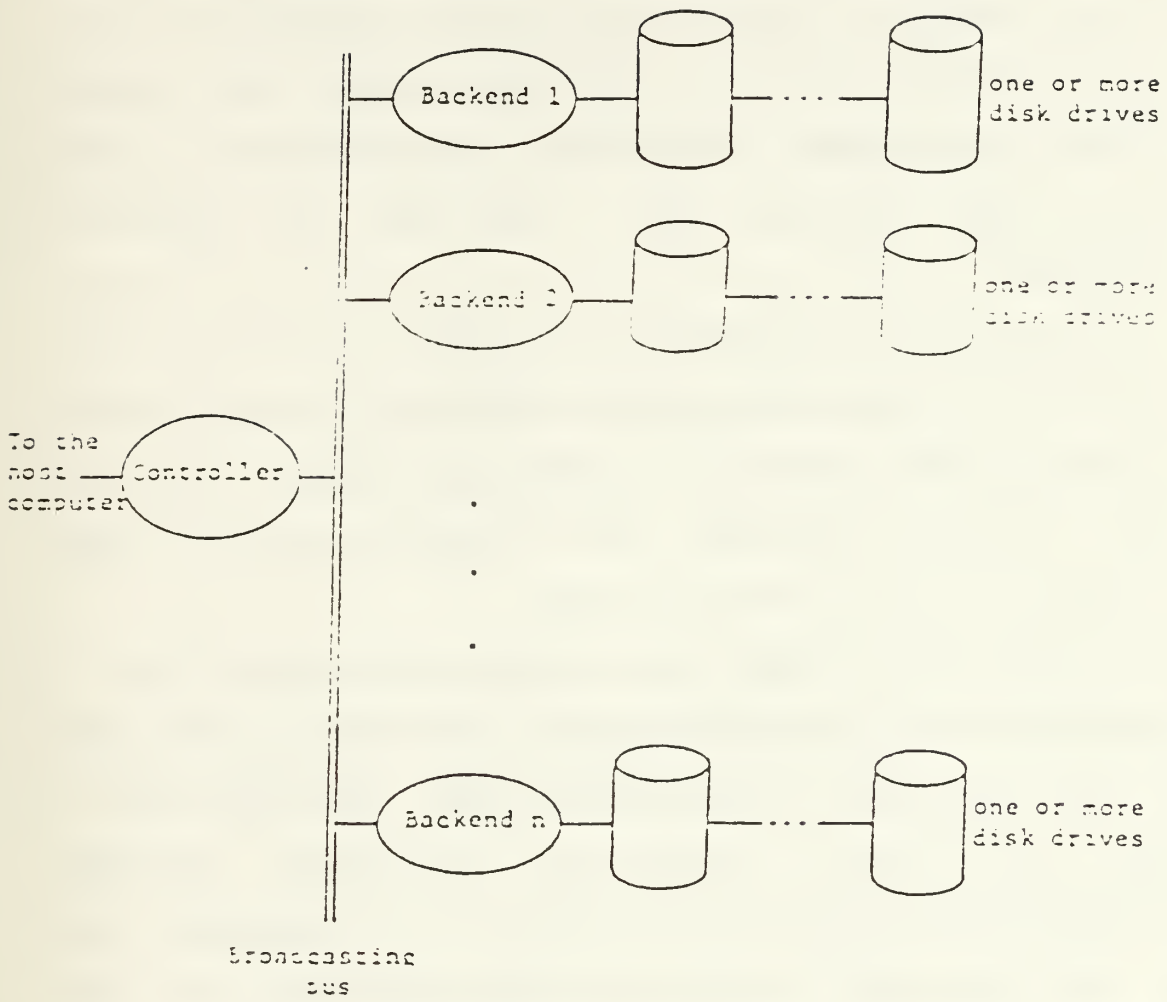

Figure 2.1. The MDBS Hardware Organization

(1) The throughput improvement is proportional to the number of backends. That is, if the number of backends is doubled, it should be possible to nearly double the size of the database without affecting the throughput.

(2) The response time is inversely proportional to the number of backends. It should be possible to nearly halve the average response time by doubling the number of backends.

(3) The system is extensible for capacity growth and/or performance improvement. By extensibility, we mean that an upgrade of the system can be made with no modification to the existing hardware and software, and no major disruption of the system activity.

To meet the MDBS design goals, the controller is implemented with the following goals. The amount of the work that the controller should perform must be minimized in order to avoid controller bottleneck problems. Communication between the controller and the backends must also be minimized in order to avoid bus contention. As a consequence of the controller implementation goals, the backends should do most of the work. Further, the communication among the backends must be minimized.

B. THE MDBS SOFTWARE ARCHITECTURE

MDBS is designed to provide for database growth and performance enhancement by the addition of identical backends and their disks. The software architecture does not require the development of new software when a backend is added. In other words, the existing software supports many backends as well as a few backends. The software architecture allows replication of the existing software for the new backends added for expansion. No new software is developed. Reconfiguration is simple, and does not require extensive system regeneration. The software architecture of MDBS is shown in Figure 2.2. For more detail, refer to [Ref. 1, 2, 3].

The software architecture also takes full advantage of the parallelism in the hardware architecture. The software of the backends supports parallel processing of the database. There are three primary features which support this parallelism. The first is the method by which the database is distributed over the disk drives of the backends.

The data model chosen for the system is the attribute-based data model [Ref. 1]. In MDBS the database consists of files of records. Each record is a collection of keywords , optionally followed by a record body. A keyword is made up of an attribute-value pair. A record body is string of characters not used by MDBS for search purposes. In

particular, the first attribute-value pair of each record of a file consists of the attribute FILE and the file name as its value. For performance reasons, records are logically grouped into clusters based on the attribute values and attribute value ranges in the records. These values and value ranges are called descriptors. At database creation time, the database creator specifies a number of descriptors. These descriptors are called as clustering descriptors that are used for forming clusters of records. An attribute that appears in a descriptor is called a directory attribute. For the purposes of clustering, only those keywords of the records which contain directory attributes are considered. Such keywords of the record are termed directory keywords.

This concept of clusters contributes to parallel processing in the following manner. The distribution of data across the backends is based on the concept of clusters. The records of a cluster are distributed accross the backends according to the distribution algorithm proposed in [Ref. 1]. Therefore, each backend has a part of the cluster. Thus, each backend may access a portion of the data required by a request. All backends can work and access their portions in parallel.

The second feature of the software architecture which exploits the paralellism is the way in which directory data is managed. Every backend has its own copy of the clusters.

The search for the descriptors related to a request can thus be shared by all of the backends.

The third feature which supports paralellism is the method used for scheduling requests and controlling concurrent access to the database and the directory data. Each backend keeps a request queue. Requests are scheduled independently, as resources become available. Concurrency is maintained separately at each backend with a locking algorithm. Thus, the backends work independently and in parallel. In exploring alternatives for the sort and join operations, we will preserve this idea of independent, parallel processing in the backends.

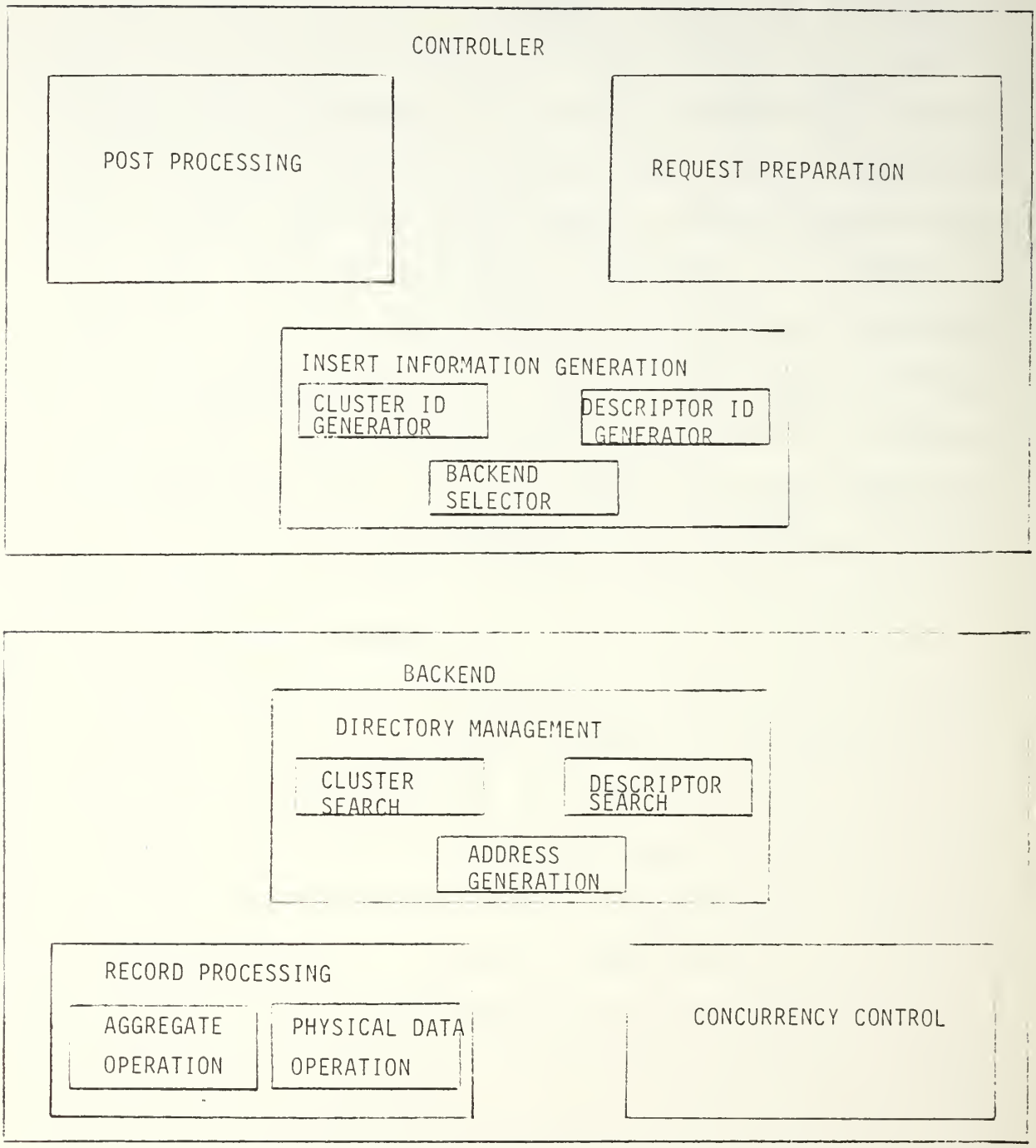

Figure 2.2. MDBS Software Architecture

III. ALTERNATIVES FOR DISTRIBUTING THE SORTING FUNCTION

When considering alternatives for distributing the sorting function among the processors of MDBS, we must consider both the hardware and software architectures. The hardware and software architectures, as explained in Chapter II, are designed for distributing the functionality of the database management operations across the backends. We must select an alternative that exploits the inherent parallelism of the architecture. The architecture of MDBS dictates minimal controller function, minimal message traffic, and identical software for the backends. The alternatives which we recommend should be consistent with the dictations made on the existing hardware and software architectures.

We will consider the complexity of the sort function to include only the overhead incurred by adding an ordering specification to a RETRIEVE request, the time required to retrieve records is not considered. We will develop expressions which represent the CPU activity, expressions which represent the I/O activity, and expressions which represent the communication activity on the bus, i.e., the computing complexity , the access complexity , and the communication complexity , respectively.

When analyzing complexity for functions distributed accross the backends, remember that the backends are working

parallel. The result of this distribution of work across B backends operating in parallel is that the linear complexity, the sum of the work done at all B backends, is reduced to an effective complexity, the work required at the one backend which does the most work. Assumption 5 is that the number of blocks to be sorted is evenly distributed across the backends. Therefore, since each backend will do an equal amount of work, the effective complexity is equal to the complexity at any one backend.

A. ASSUMPTIONS

In each case, we will analyze the worst-case complexity of the current alternative. In order to simplify the analysis, we make the following assumptions.

(1) Internal sorting only is considered, due to memory limitations. The backends are currently 16-bit minicomputers with a fixed, 32 K-byte address space. Therefore, memory limitation is a real problem.

(2) All records in a block are to be sorted (i.e., selection of records is performed by record processing before sorting).

(3) Sorting is block-by-block (i.e., a block of records selected by the record processing function is passed to the sorting function, where they are sorted and stored in the secondary storage for merging).

(4) Merge is 2-way. This is the simplest case. We will consider K-way merge in Chapter V.

(5) The number of blocks to be sorted is evenly distributed across the backends (i.e., if there are M blocks to be sorted and B backends, then each backend sorts M/B blocks).

(6) Some sorting algorithms of the order (r*log r), where r is the number of records, will be used.

(7) Records are sorted on a single concatenated key (i.e., only a single execution).

(8) The time to send a block of data across the broadcast bus is an average time, which will be represented as a constant.

(9) The time to read (or write) a block of data from (or to) the disk is an average time, which will be represented as a constant, and is the same for the controller and the backends.

(10) The CPU time required for a comparison operation is the same at the controller and at the backends.

B. NOTATION

In analyzing the time complexity, we will deal with variables which represent the number of backends, the number of records to be sorted, the number of records in a block, etc. We will also deal with certain constants. For example, according to assumption 8 above, there is some constant

which represents the time required to send a block of data across the broadcast bus. For uniformity, we define the following variables and constants to be used throughout the analysis.

(1) B = the number of backends in the system.

(2) N = total number of records to be sorted for a particular request.

(3) r = the number of records in a block.

(4) b = the number of blocks to be sorted at the backend. Note that according to assumption 5 , b=N/(B*r). To simplify the analysis, we will assume that b is a power of 2.

(5) log : stands for logarithm to the base 2 unless otherwise noted.

C. SYNTAX FOR THE SORT FUNCTION

The syntax of a retrieve request in MDBS is as follows.

RETRIEVE Query Target-list [BY attribute][WITH pointer]

That is, it consists of five parts. The first part is the name of the request. The second part is a query which identifies the portion of the database to be retrieved. The Target-list is a list of elements. Eack element is either an attribute or an aggregate operator to be performed on an

attribute. The fourth part of the request, BY clause, is optional. It describes the whole alternative of the attribute such that BY DEPT means every department in the database. The fifth part of the request, WITH pointer, is also optional which specifies whether pointers to the retrieved records must be returned to the user or user program for later use in an update request which is out of our concerns for sort function.

To perform the sort function, we first need to retrieve the records that are relevant to the user request. Therefore, modified retrieve request can be used as a syntax for the sort function.

With modified RETRIEVE request, we may consider two different alternatives for a syntax to implement the join function in MDBS:

1) RETRIEVE Query Target-list (ORDER_BY (Attribute_list_1))

2) RETRIEVE Query (ORDER_BY (Attribute_list_1),
(Attribute_list_2))

In both alternatives, the first two parts are the same as in regular retrieve request. In the first alternative, Target-list clause consists of the attribute names with which the result of the sort function is given to the user. ORDER_BY clause defines the function to be performed on the retrieved records. Attribute_list_1 defines the list of

attributes' names with which the retrieved records are sorted. If there are more than one attribute name in the Attribute_list_1, then it be assumed that the order of the attributes in attribute_list_1 gives the order of implementation of consequitive sort function on the retrieved records. Attribute_list_1 may contain either Directory Attribute(s) or non-directory attributes or both. The important point is that each attribute in attribute_list_1 must be an attribute that the records retrieved from database obtain it.

In the second alternative, Atribute_list_1 includes the attribute names with which the record are sorted. The order of performing the sort function on the records is again the same as the order of the attributes given in attribute_list_1.

Attribute_list_2 includes the attribute names with which the result of the sort function is given to the user. In other words, it can be thought as a target-list.

IV. THE ISSUE OF DISTRIBUTION OF FUNCTIONALITY

In analyzing the distribution of function, we will assume that the sort function consists of two phases: the internal sort phase and the merge phase. Because of main memory limitations, we require that the records first be sorted block-by-block. The sorted blocks are stored in temporary storage in the secondary memory. This is done by the internal sort phase. Sorted blocks will then be accessed from the secondary storage and merged. This is done by the merge phase. The time complexity of these two processes will be shown separately. At the end of the analysis of each alternative, the total time complexity will be given.

We will consider three alternatives regarding distribution of function through the system. Since MDBS consists of two type of functional units, namely the controller and the backends, the possible distributions of functionality are the following;

A. The controller performs the sort function.

B. The backends perform the sort function.

C. The controller and the backends share the sort function.

We will analyze these three alternatives in detail in the following sections.

A. THE CONTROLLER PERFORMS THE SORT FUNCTION

In this alternative, the backends perform no additional functions. All of the sorting is done at the controller. The backends perform the selection, projection, and aggregation operations specified in the RETRIEVE request, and forward the result records to the controller. The controller accumulates the result records from all of the backends, and sorts them in the order specified in the RETRIEVE request before forwarding them to the requester.

There is no change in the functionality of the backends. Therefore, no modification of the software of the backends is required. However, at least two processes in the controller will require modification. First, the request processing process must be augmented to recognize the ordering specification of the request, and to forward the ordering specification to the post-processing process. The post-processing process must be augmented to recognize that sorting is required, and to accumulate and sort result records for a request according to the proper ordering specification.

First, we assume that all blocks for a query have been accumulated and stored in the secondary storage of the controller. The controller will have (B*b) blocks to sort.

The internal sort phase for each block will require O(r*logr) time, where there are r records per block. The total computing complexity for the internal sort phase time is, then,

$$O(B*b*r*\log r).$$

2*B*b accesses to the secondary storage are required during the internal sort phase. So, the access complexity of the internal sort phase is

$$O(B*b).$$

Since there are (B*b) blocks at the controller, log(B*b) will be the number of passes over data for merging. Each pass will require (B*b*r - 1) comparisons. So, the computing complexity for the merge phase will be (log(B*b)*(B*b*r-1)), which is in

$$O(B*b*r*\lceil \log(B*b) \rceil).$$

2*B*b*log (B*b) accesses to the secondary storage are required for merging, so the access complexity of the merge phase is

$$O(B*b*\lceil \log (B*b) \rceil).$$

Therefore, the worst-case computing complexity for the sort function is

$$O(B*b*r*(\log (B*b*r)), \text{or}$$

$$O(N * \log N),$$

and the access complexity is

$$O(B*b*\log(B*b)), \text{ or}$$

$$O((N/r)*(\log(N/r)).$$

In this case, since all sorting and merging is done by one processor, the controller, the effective complexity and the linear complexity are the same.

B. THE BACKENDS PERFORM THE SORT FUNCTION

Here we consider two strategies. In the first, all of the backends share the internal sort phase, and the merge phase is performed by one or two backends. In the second, each backend sorts and merges the blocks of data resident at that backend. The backends then share the work of merging with B/2 backends performing the first partial merge, B/4 backends performing the next partial merge, etc. Let us examine each of these strategies in datail.

1. All Backends Sort, and One or Two Backends Merge

In this alternative all backends perform the internal sort phase individually. After the internal sort phase is complete, one or two predetermined backends complete the process by merging all of the sorted blocks.

So each backend sorts b blocks of r records. The computing complexity of the internal sort phase at one

backend is

$$O(b*r*\log r),$$

and 2*b accesses to the secondary storage are required, so the access complexity of the internal sort phase is

$$O(b).$$

This is the effective complexity for sorting. Since the work of sorting is shared among the backends, we use this effective complexity in our analysis.

Next, the sorted blocks of records must be transmitted along the broadcast bus to the one or two backends which will perform the merge phase. Let us take the case where one backend does all the merging. Then, if there are B backends, (B-1)*b blocks must be transmitted. The communication complexity is

$$O(B*b).$$

Also, B*b accesses to the secondary storage are required to store the transmitted blocks at the backend assigned to perform the merge phase. This requires the access complexity

$$O(B*b).$$

The backend selected to perform the merging now has (B*b) blocks. Merging (B*b) blocks at the backend requires the time of (B*b*r-1)*log (B*b). The computing complexity is

$$O(B*b*r *\lceil \log(B*b) \rceil) ,$$

since 2*B*b*log (B*b) accesses to the secondary storage are required, the access complexity of the merge phase at this backend is

$$O(B*b*\lceil \log (B*b) \rceil).$$

Therefore, the total computing complexity for the sort function is

$$O(b*r*(\log r + B*\lceil \log (B*b) \rceil)),$$

the access complexity is

$$O(b*B*\log (B*b)),$$

and the communication complexity is O(B*b).

2. All Backends Sort Separately and Share Merging

In this strategy, all the backends, as in the previous section, share the work of sorting. Therefore, the computing complexity of the internal sort phase is, again, the effective complexity,

$$O(b*r*\log r),$$

and the effective access complexity is O(b).

Then, each backend performs the merge phase over its own b blocks. This requires the computing complexity of

$$O(b*r*\lceil \log b \rceil)$$

and the access complexity of

$$O(b*\lceil \log b \rceil).$$

Next, the merge phase is shared by the backends in the manner shown in Figure 4.1. First, B/2 backends perform a merge pass, each merging 2*b blocks. Then B/4 backends perform a merge pass, each merging 4*b blocks. This process is repeated log B times. Now let us look at the computing complexity of the merge phase step by step.

1. step	(2*b*r - 1)* log (2)
2. step	(4*b*r - 1)* log (2)
3. step	(8*b*r - 1)* log (2)
4. step	(16*b*r - 1)* log (2)
5. step	(32*b*r - 1)* log (2)
.	. .
.	. .
.	. .
$\lceil \log B \rceil$ step	$(2^{\lceil \log B \rceil} * b*r - 1)* \log (2)$

The expression for the computing complexity of the merge phase, then, as derived from the above, is

$$O(b*r*(2^{\lceil \log B \rceil})).$$

Again, this is the effective complexity.

At each step, each target backend first stores the blocks transmitted from their neighbor backends before the merge phase starts. This requires the access complexity

$$O(b*2^{\lceil \log B \rceil}).$$

Since the merge phase is performed in log B steps, the access complexity of the merge phase is derived as the following.

1. step (2*b)*log(2)
2. step (4*b)*log(2)
3. step (8*b)*log(2)

. . .

log B step $(2^{\lceil \log B \rceil}$*b)*log(2)

Therefore, the effective access complexity of sharing the merge phase for this alternative is

$$O(b*(2^{\lceil \log B \rceil})).$$

At each step, one half of the total number of blocks must be transmitted over the broadcast bus to the target backends for the next step. Since there are log B steps, the communication complexity between the backends is

$$O(B*b*\lceil \log B \rceil).$$

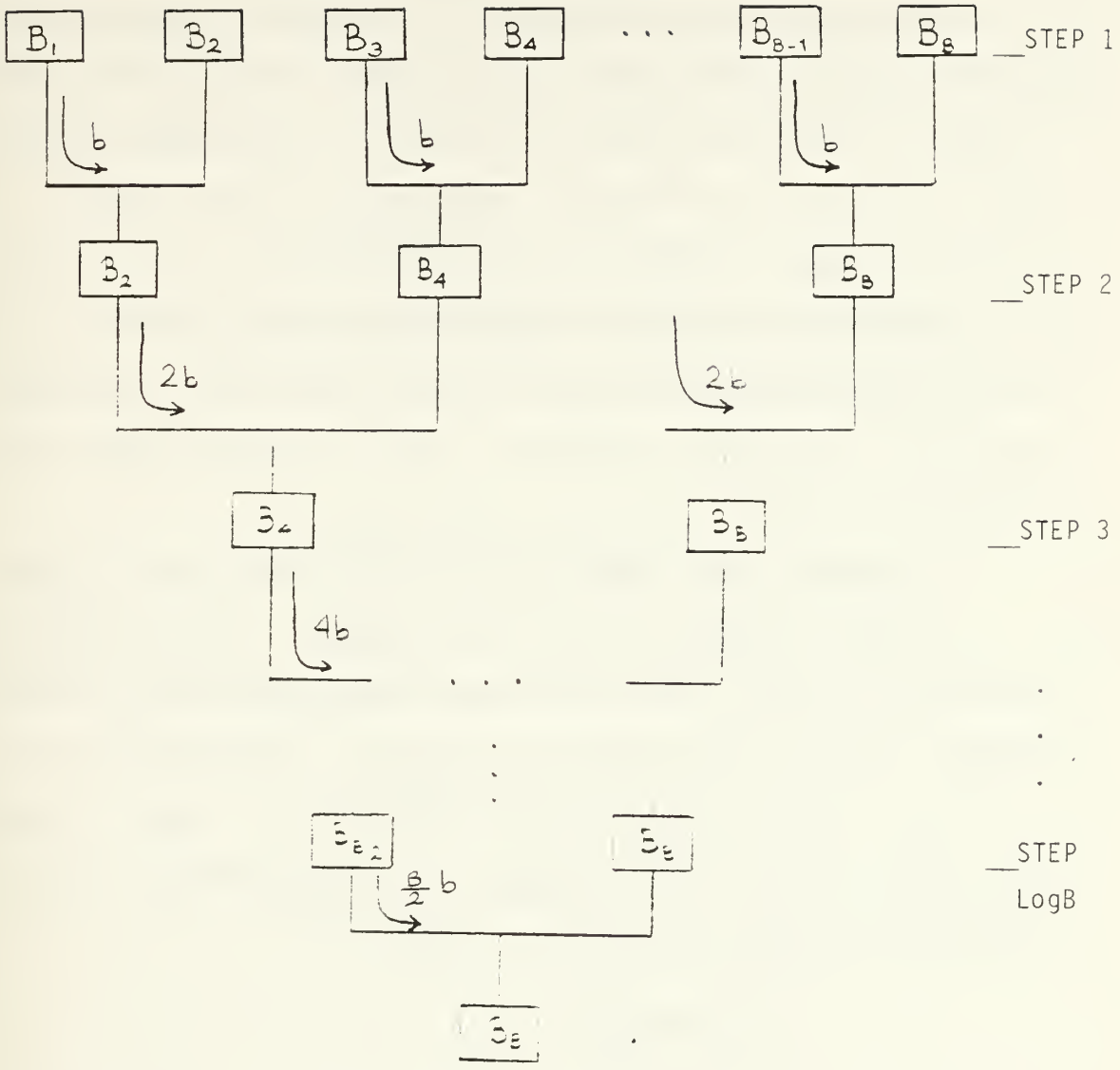

Figure 4.1. Performing the Sort Function Step-by-Step at the Backends

C. THE CONTROLLER AND THE BACKENDS SHARE THE SORT FUNCTION

We examine two strategies for distributing the sorting function between the controller and the backends. The first strategy is that the backends perform only the internal sort phase, and the controller performs the merge phase. The second strategy is that the backends perform the internal sort phase and a partial merge, merging all of the records in the blocks stored at that backend, and the controller completes the merge process. Let us examine each of the strategies in detail.

1. Backends Sort Block-by-Block and Controller Merges

Every backend performs the internal sort phase on its part of the file. Each block is sorted and forwarded directly to the controller for merging. The time complexity for the internal sorting of a block is O(r log r), where there are r records in a block. The effective computing complexity of the internal sort phase is

$$O(b*r*\log r),$$

where there are b blocks per backend. 2*b accesses to the secondary storage are required, so the effective access complexity is

$$O(b).$$

The sorted blocks are sent to the controller via the broadcast bus. This communication cost is included in the

cost of a RETRIEVE operation, and is not an overhead cost for sorting. However, the controller must store (B*b) blocks before the merge phase starts. This requires the access complexity

$$O(B*b).$$

The controller now has B*b blocks to be merged. The computing complexity for a 2-way merge is (log(B*b)*(B*b*r-1)), which is

$$O(B*b*r* \lceil \log (B*b) \rceil).$$

2*B*b*log(B*b) accesses to the secondary storage are required, so the access complexity for the merge phase is

$$O(B*b* \lceil \log (B*b) \rceil).$$

So,the computing complexity for this alternative is

$$O(b*r*\log r + B*b*r* \lceil \log(B*b) \rceil) \text{ or}$$
$$O(b*r*(\log r + \lceil \log(B*b) \rceil)),$$

and the access complexity is

$$O(b*B* \lceil \log(B*b) \rceil).$$

2. The Backends Sort and Perform a Partial Merge, and the Controller Performs the Final Merge

In this case every backend sorts its part of the requested file, and the controller merges those partially

sorted file parts being sent from every backend. Since the backends perform the internal sort phase block-by-block, the effective computing complexity of internal sort phase is

$$O(b*r*\log r).$$

Assuming that every internally sorted block is stored back into the secondary storage, the access complexity is

$$O(b).$$

Now each backend merges the sorted blocks resident at that backend. The number of passes over the data required for the merge phase is log b. Therefore, the effective computing complexity of merging b blocks at the backend is (b*r-1)*log b, or

$$O(b*r*\lceil \log b \rceil),$$

and the access complexity is

$$O(b*\lceil \log b \rceil).$$

So, the computing complexity for the internal sort and merge phases at the backends is

$$O(b*r*(\log r + \lceil \log b \rceil)),$$

and the access complexity is

$$O(b*\lceil \log b \rceil).$$

Communication of these blocks to the controller is, again, not a part of the sorting cost. However, since the transmitted blocks are to be stored before the merge phase starts, this requires the access complexity

$$O(B*b).$$

The controller, now, will have B runs of sorted records to be merged. The logarithmic value of the number of backends gives the number of passes over data, log B. So the computing complexity of the merge phase at the controller is

$$O(B*b*r*\lceil \log B \rceil),$$

and the access complexity is

$$O(B*b* \lceil \log B \rceil).$$

D. EVALUATING THE ALTERNATIVES

In the previous sections we have presented five alternative distributions of funtionality between the controller and the backends. In this section we will analyze the tradeoffs of the alternatives. Table 1 summarizes the computing complexities of the internal sort and the merge phases, the access complexity, and the communication complexity for all five alternatives.

ALTERNATIVE	COMPUTING COMPLEXITY				ACCESS COMPLEXITY		COMMUNICATION COMPLEXITY	
	INTERNAL SORT PHASE		MERGE PHASE					
	AT THE Bs	AT THE C	AT THE Bs	AT THE C	AT THE Bs	AT THE C	AMONG THE Bs	AMONG THE C & Bs
A	—	$O(B*b*r*\log r)$	—	$O(B*b*r*\log B*b)$	—	$O(B*b*\log(B*b))$	—	—
B.1	$O(b*r*\log r)$	—	$O(B*b*r*\log B*b)$	—	$(b*b*\log B*b)$	—	$O(B*b)$	—
B.2	$O(b*r*\log r)$	—	$O(b*r*(2^{\lceil \log B \rceil} + \log b))$	—	$O(b*(2^{\lceil \log B \rceil} + \log b))$	—	$O(B*b*\log B)$	—
C.1	$O(b*r*\log r)$	—	—	$O(b*b*r*\log B*b)$	$O(b)$	$O(B*b*\log B*b)$	—	—
C.2	$O(b*r*\log r)$	—	$O(b*r*\log b)$	$O(b*b*r*\log b)$	$O(b*\log b)$	$O(b*b*\log b)$	—	—

Bs: Backends, C: Controller

Table 1 The Time Complexities for the Alternative Distributions of the Sort Function

Alternative A represents the distribution of function presented in Section A of this chapter. The controller performs all of the sorting and all of the merging. Alternative B.1 represents the distribution presented in section B.1 of this chapter. All of the backends perform the sorting , and one or two backends perform the merging of the sorted blocks. Alternative B.2 represents the distribution of function presented in Section B.2 of this chapter. All backends perform the sorting and share the merging.

Alternative C.1 represents the distribution of function presented in Section C.1 of this chapter. All the backends perform the sorting and the controller performs the merging. Finally, alternative C.2 represents the distribution of function presented in Section C.2 in this chapter. Backends sort and perform a partial merge, and the controller performs final merge.

The complexity formulas of those both accesses to the secondary storage and block transmission are given only for the additional accesses or transmissions necessary to complete the sort function. In other words, accesses to the secondary storage to retrieve the records in order to perform selection and projection before the sort function starts, and transmission of the blocks from the backends to the controller are not included. In general, each alternative, except A, has the same time complexity with

regard to the internal sort phase. Therefore, we will focus on the other columns in comparing the alternatives.

First, let us examine alternative A, where the controller performs all sorting and merging. The computing complexity is O(B*b*r* log(B*b)) for sorting and merging (B*b) blocks of r records. As easily seen, this alternative is contrary to the design goal of the minimizing controller function. Therefore, we will eliminate it from further considerations.

Next, let us examine alternative B.1, where all backends perform the sorting and one or two backends perform the merge. The backends perform all of the work of sorting and merging. Even though this alternative seems to meet the design goal of minimizing the controller function, it is contrary to the second design goal of minimizing the message traffic between the backends. The communocation complexity is O(B*b) for (B*b) blocks. Clearly, for queries involving a large number of blocks, the communication overhead will be high and bus congestion may result. Another disadvantage is that a single backend performs the merging. Also, when the single backend is performing the merging, it may delay the processing of other queries, thus causing a decrease in system throughput. Because of the communication overhead and the potential for decrease in throughput, we will also eliminate alternative B.1 from further consideration.

Next, we consider alternative B.2, where all backends share the sorting and the merging. The communication complexity is O(B*b*log B) for (B*b) blocks. The communication complexity increases logarithmically with B, the number of backends. Clearly, this alternative is also contrary to design goal of minimizing the message traffic between the backends. Also, the computing complexity for the merge phase and the access complexity increase exponentially by log B, where B is the number of backends. Clearly, with this alternative, increasing the number of backends will cause longer response time and decreased throughput. So, we will not consider B.2 to be a desirable distribution of function.

This leaves us with alternatives C.1 and C.2. Alternative C.1 is that the backends perform the sorting block-by-block and the controller merges all the blocks. Alternative C.2 is that the backends perform the sorting and a partial merge, and then the controller performs the final merge. Neither alternative incurs transmission overhead. Therefore, the design goal of minimizing the bus traffic is met.

In both alternatives the work of sorting and merging is shared between the backends and the controller. Alternative C.1,however,does involve more work for the controller than alternative C.2. Since the backends perform the main portion of the merge process in C.2 the controller's work is

reduced. On the other hand, the workload of the backends is greater with alternative C.2, than with alternative C.1. Let us analyze these two alternatives with respect to the design goals of minimizing the controller function and maximizing the work done by the backends in the next section.

E. COMPARISONS BETWEEN ALTERNATIVES C.1 AND C.2

In this section we will compare the two alternatives, namely C.1 and C.2. In comparing these two alternatives, we will analyze computing complexity and access complexity separately. Since the time to do one disk access is much longer than the CPU time to perform one comparison, separate analyses will be more meaningful.

As is shown in Table 1, the internal sort phase computing complexity is the same for both alternatives. However, with alternative C.2, the backends perform a part of the merge. Consider that, for both alternatives, if the number of blocks is held constant, increasing the number backends will cause the number of blocks to be sorted at one backend, b, to decrease. This decrase is linear work respect to the number of backends. Therefore the computing complexity on the backends decreases linearly with an increasing number of backends.

However, meeting the amount of work done by the controller' function is clearly less with alternative C.2

than with alternative C.1, due to the fact that the backends offload some of the work of merging from the controller. Consider the case where the total number of recors (N=B*b*r) is held costant. The computing complexity of alternative C.1 will not vary with the number of backends. For the alternative C.2, the computing complexity of the merging at the controller will increase logarithmically with the number of backends. However, the computing complexity for merging at the controller will always be less for the alternative C.2 than C.1 by a factor of (B*b*r*log b). Since b decreases as B increases, the gain will be proportionately smaller as B grows large. Clearly, however, the alternative C.2 better fits the goal of minimizing the controller function. Clearly, a substantial reduction in the controller workload will result from assigning more functionality to the backends.

Now let us examine the effect of increasing the number of backends. We will analyze the computing complexities, access complexities, and communication complexities of both alternatives. Let us examine the case that the total number of records, N=B*b*r, and the number of records per blocks, r, remaining constant, while the number of backends, B, increases.

For alternative C.1, the total computing complexity is

$b*r*\log r + B*b*r*\log(B*b)$, or

$N*(\log N - ((B-1)/B)*\log r)$, since $b=N/(B*r)$.

This obviously yields decreasing results for increasing values of B. This reduction,however,will have minor effect on the result of the computing complexity. Further, the reduction can be ignored for large N values.

For alternative C.2, the total computing complexity is

$b*r*\log r + b*r*\log b + B*b*r*\log B$, or

$N*((1/B)*\log N + ((B-1)/B)*\log B)$, since $b=N/(B*r)$.

$(1/B)*\log N$ obviously decreases with the increasing B. However, $((B-1)/B)*\log B$ increases for increasing B values. There is some breakpoint for where the effect of the decreasing term has more effect than the increasing term. Let us assume that we double the number of backends. The difference in the total complexities between the case that the backends are not doubled and the case that do double is

$N*((1/(2*B))*\log N -(1/(2*B))*\log B -((2*B-1)/(2*B)))$.

As long as the condition, $\log N > \log B + (2*B-1)$, holds, the total complexities will be reduced. So, if the condition $N > B*2^{2B-1}$ holds after doubling the number of backends, then the computing complexity decreases.

. Now, let us examine the access complexity. For the alternative C.1, the total access complexity is

b + B*b*log(B*b), or

N*((1/(B*r)) + log(N/r)), since b=N/(B*b).

Clearly, this complexity decrease as B increases. However, the decrease still has minor effect, especially for a large N.

The access complexity for alternative C.2 is

b*log b + B*b*log B, or

(N/r)*((1/B)*(log N - log r) + ((B-1)/B)*log B).

Again (1/B)*(log N - log r) decreases , but ((B-1)/B)*log B increases as B increases. Let us again assume that we double the number of backends. The difference in complexities going from B backends to 2*B backends is

(N/r)*((1/(2*B))*(log N -log B -log r -2*B + 1)).

As long as the condition, log N > log B + log r + 2*B-1, holds, the total access complexity decreases. So, if the condition $(N/r) > B^{*2^{2B-1}}$ holds after doubling the number of backends, then the total access complexity decreases.

Figure 4.2 illustrates the computing and access complexities for both alternatives for $N=2^{13}$ and $N=2^{15}$ and r=64 as B increases. As is easily seen, alternative C.2 is always better than alternative C.1 for meeting the design goals of MDBS.

F. RECOMMENDED DISTRIBUTION OF FUNCTIONALITY

In the previous sections of this chapter we have analyzed the alternatives of the distribution of the functionality and shown the tradeoffs and the advantages of each one. Briefly, alternatives A, B.1, and B.2, are contrary to the design and implementation goals of MDBS. The other two alternatives, C.1 and C.2, are pertinent for our concerns.

At each comparison for alternatives C.1 and C.2 in the previous section, we have shown that C.2 is better alternative for large number of records. Therefore, we recommend that the functionality be distributed in the following manner: the backends perform the sorting and partial merge , and the controller performs the final merge.

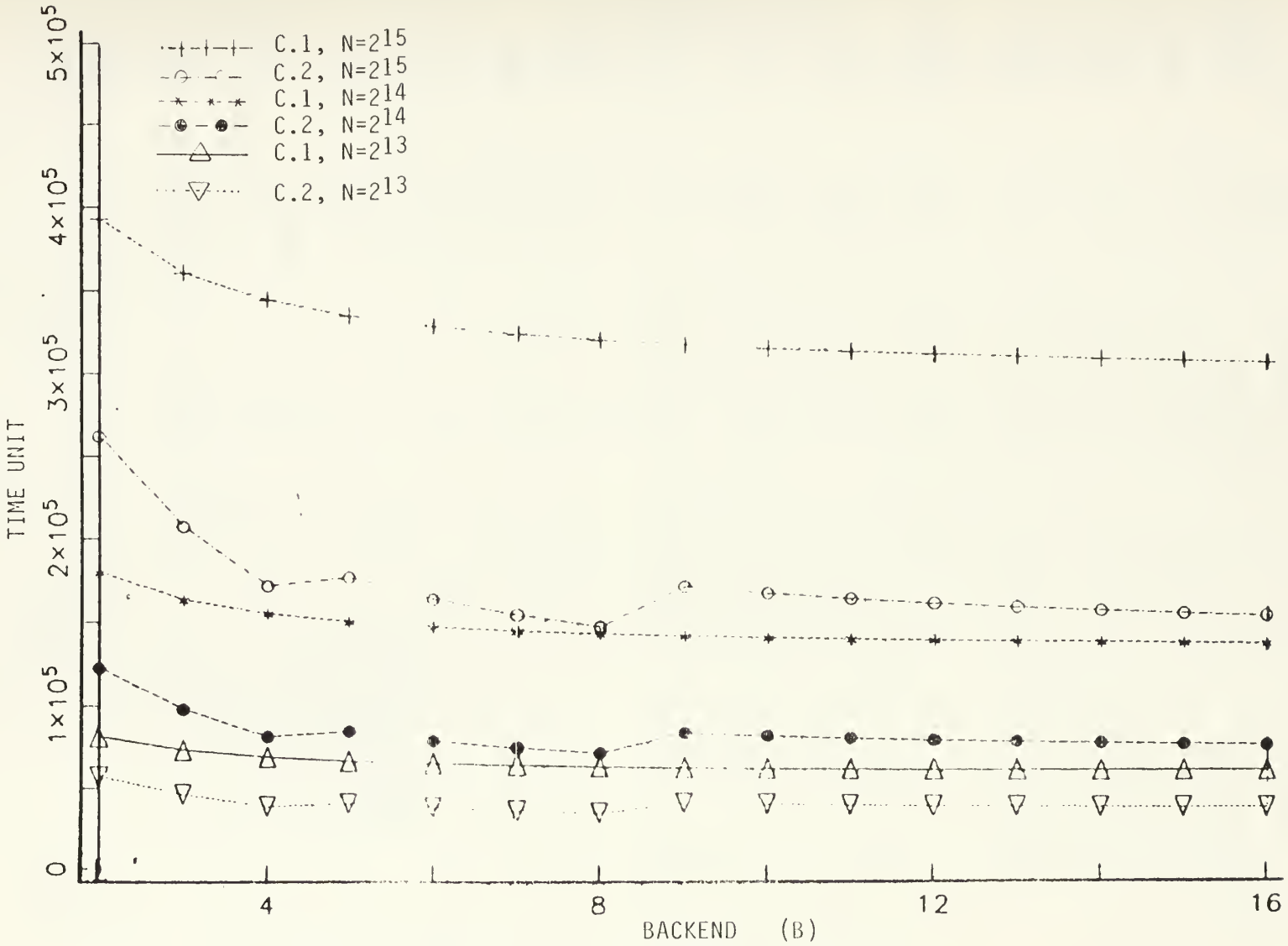

Figure 4.2. Computing Complexity for Alternatives C.1 and C.2

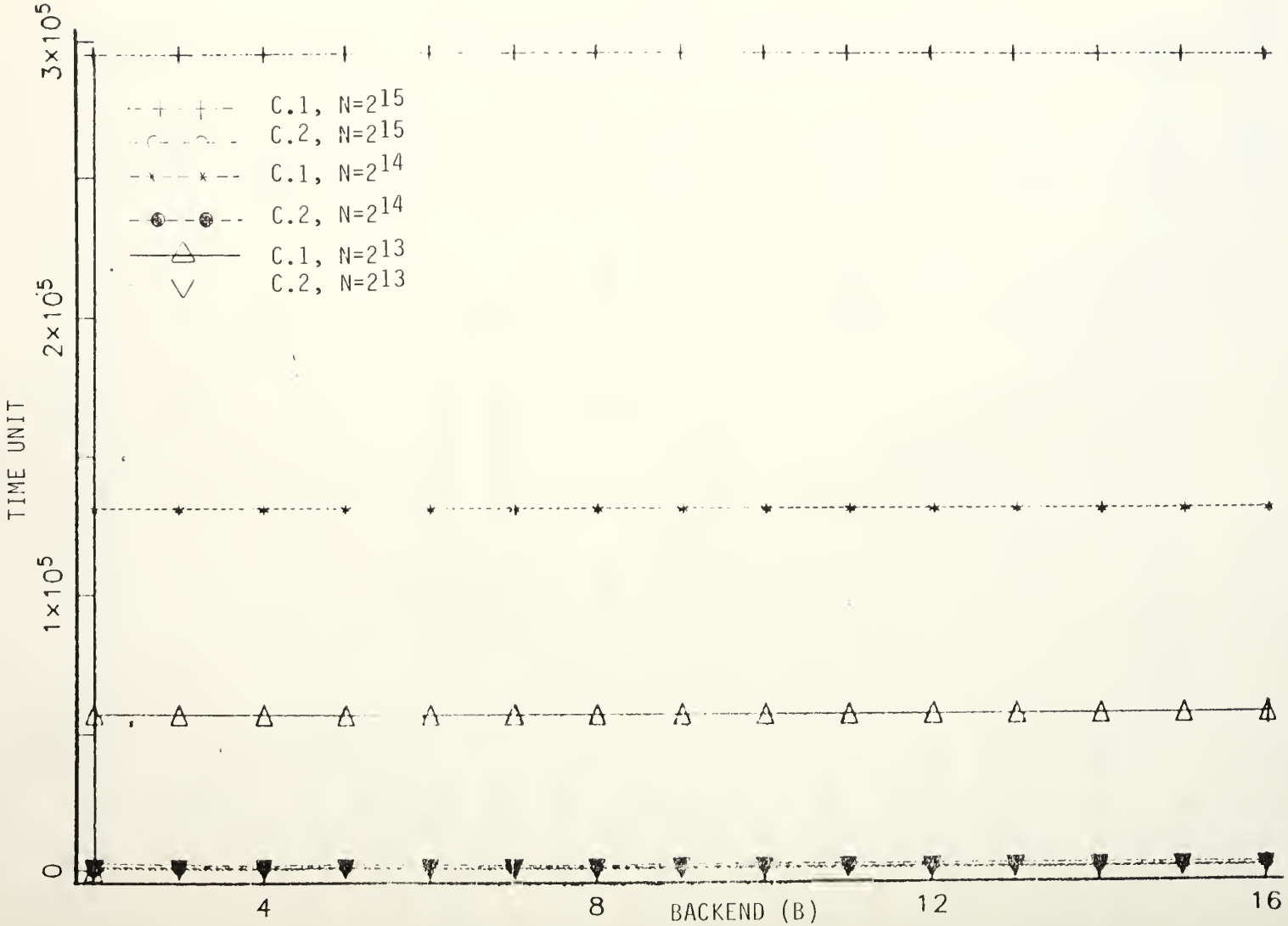

Figure 4.3. Access Complexity for Alternatives C.1 and C.2

V. DIFFERENT ALGORITHMS FOR THE SORT AND MERGE PHASES

In our previous analyses, we made the assumption that records are sorted one block at a time, using some well-known sorting algorithm with time complexity of O(r*log r), where r is the number of records in a block. In this chapter, we will examine the effect of sorting records n blocks at a time, and the effect of using a k-way merge.

A. SORTING WITH n BLOCKS AT A TIME

In this case, we sort n blocks at a time. There are two cases to consider. First, if the sorted blocks are stored back into the backend's secondary storage, our analysis will be the same as the previous one, except that the coefficients of the computing complexity formulae will be proportional to n.

The computing complexity for n-blocks-at-a-time sorting is O(n*r*log n*r). This process will be repeated b/n times. Therefore, the effective computing complexity is O(b*r*log n*r). The access complexity remains the same, O(b). Since there will be b/n runs to be merged, the number of passes over data becomes log (b/n). Computing complexity for merge phase, then, will be O(b*r*log(b/n)). Access complexity for the merge phase is O(b*log(b/n).

Table 2 summarizes time complexities for both block-by-block and n-block-at-a-time algorithms. As is easily seen, the computing complexity for sorting n-block-at-a-time algorithm is (b*r*log n) times that for sorting block-by-block. However, the computing complexity for merging is less by (b*r*log n), and the access complexity is less by (b*log n).

Figure 5.1 shows the effect of increasing the number of backends on the throughput of the CPU when sorting n-block-at-a-time. The x-axis shows the number of backends, and y-axis shows the computing complexity of sorting and merging at the backends. The y-axis is shown with log scale. These values were derived as follows. The complexity formulae are expressed in terms of N, the total number of records to be sorted, and B, the number of backends. The computing complexity required at the backends is

$$(b*r*\log(n*r)) + (b*r*\lceil \log(b/n) \rceil) \text{ or}$$

$$(N/B)*(\log N - \log B), \text{ since } b=N/(B*r).$$

Using various values of N, varying B from 2 to 16, we arrive at the curves shown in Figure 5.1.

ALTERNATIVE	COMPUTING COMPLEXITY				ACCESS COMPLEXITY		COMMUNICATION COMPLEXITY	
	INTERNAL SORT PHASE		MERGE PHASE					
	AT THE Bs	AT THE C	AT THE Bs	AT THE C	AT THE Bs	AT THE C	AMONG THE Bs	AMONG THE C & Bs
BLOCK-BY-BLOCK	$O(b*r*\log r)$	—	$O(b*r*\log b)$	$O(B*b*r*\log B)$	$O(b*\log b)$	$O(B*b*\log B)$	—	—
N-BLOCK AT-A-TIME	$O(b*r*\log(n*r))$	—	$O(b*r*\log\frac{b}{n})$	$O(B*b*r*\log B)$	$O(b*\log\frac{b}{n})$	$O(B*b*\log B)$	—	—

Bs: Backends, C: Controller

Table 2 Algorithm Block-by-Block vs. Algorithm N-Block At-a-Time

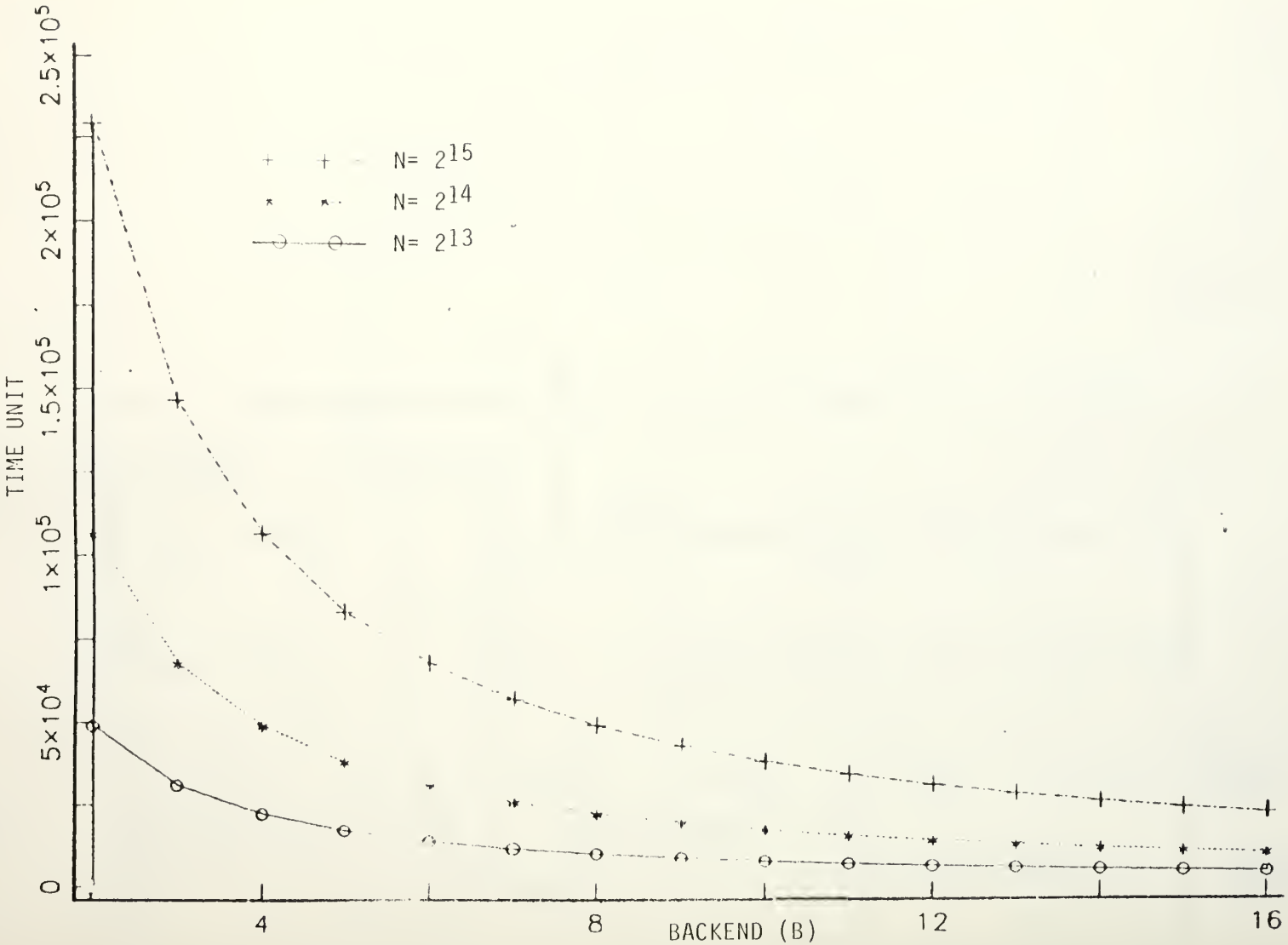

Figure 5.1. Computing Complexity of n-block Sorting with Alternative C.2

B. THE K-WAY MERGE

Up to this point, we have utilized 2-way merge process for our analyses. As we recall, the time complexity of merge is dominated by the number of runs, i.e., the number of blocks which are already internally sorted. The logarithmic value of the number of the runs gives the number of passes over data. In 2-way merge, the number of passes is the logarithm base 2 of the number of runs. If we increase order of the merge to k, for k-way merge, the number of passes will be the logarithm to the base k of the number of runs.

Let us examine what we gain with this reduced number of passes. We assume that all runs are of equal length. The notation used is:

R = the number of runs = b/n
log x = logarithm base 2 of x
LOG x = logarithm base k of x

First of all, our access time will be reduced.

In 2-way merge, access complexity is :

$$O(b* \lceil \log R \rceil).$$

In k-way merge, access complexity is:

$$O(b* \lceil LOG R \rceil).$$

Figure 5.2 gives us some information about the reduction in access complexity for a fixed number of R, and a fixed number of blocks, b, as k increases. The x-axis shows k, where k is the number of blocks merged at one time. The

y-axis is scaled as maximum 1, where 1 is the access complexity for the 2-way merge. The ratio of k-way merge access complexity to the 2-way merge complexity is graphed here. For instance, the access complexity for a 4-way merge is one half of that for a 2-way merge.

On the other hand,of course, increasing k will increase the computing complexity, or time that is necessary to compare the values. In the 2-way merge, the computing complexity is

$$O(b*r* \lceil \log R \rceil).$$

In k_way merge, computing complexity is

$$O(b*r*(k-1)* \lceil LOG\ R \rceil) \text{ or}$$

$$O(b*r*(k-1)* \lceil (\log R / \log k) \rceil) ,$$

since LOG(R)= log(R) / log k.

Figure 5.3 shows the increase in computing complexity with regard to k for a fixed R. The y-axis for this formula is scaled starting with minimum 1, where 1 represents the computing complexity for a 2-way merge at the backend. The ratio of k-way merge computing complexity to the 2-way merge computing complexity is graphed here. For instance, the 4-way merge computing complexity is 1.5 times that of the 2-way merge computing complexity.

As seen from Figure 5.4, the access complexity reduces rapidly up to 40% of the 2-way merge complexity at k=6. However, at this point the computing complexity has doubled. After this point, k>6, the reduction in access complexity becomes negligible relative to the increasing computing complexity. Therefore, we can take the point, k=6, as an implementation point for the degree of the merge.

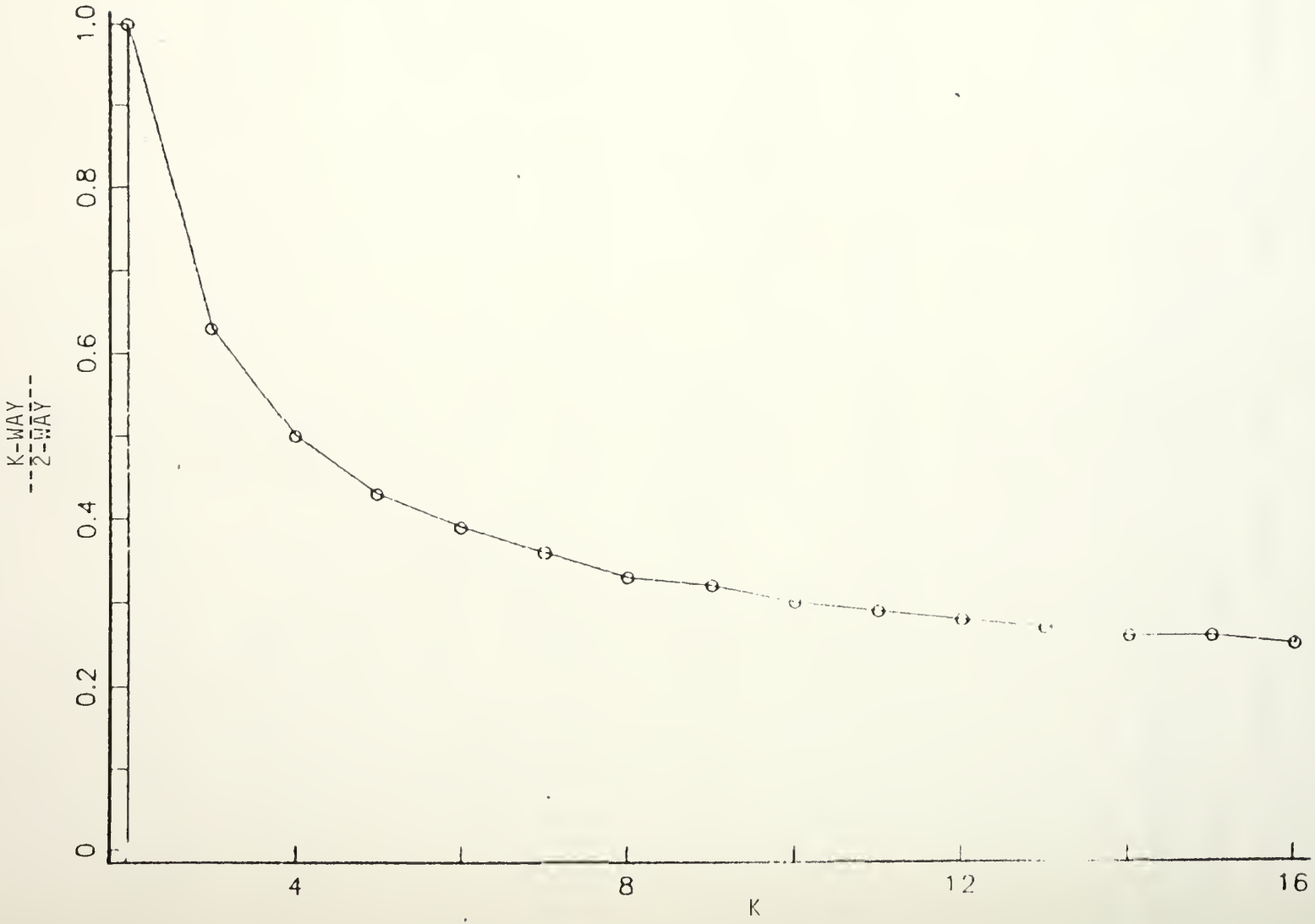

Figure 5.2. Access Complexity for Merging with various K

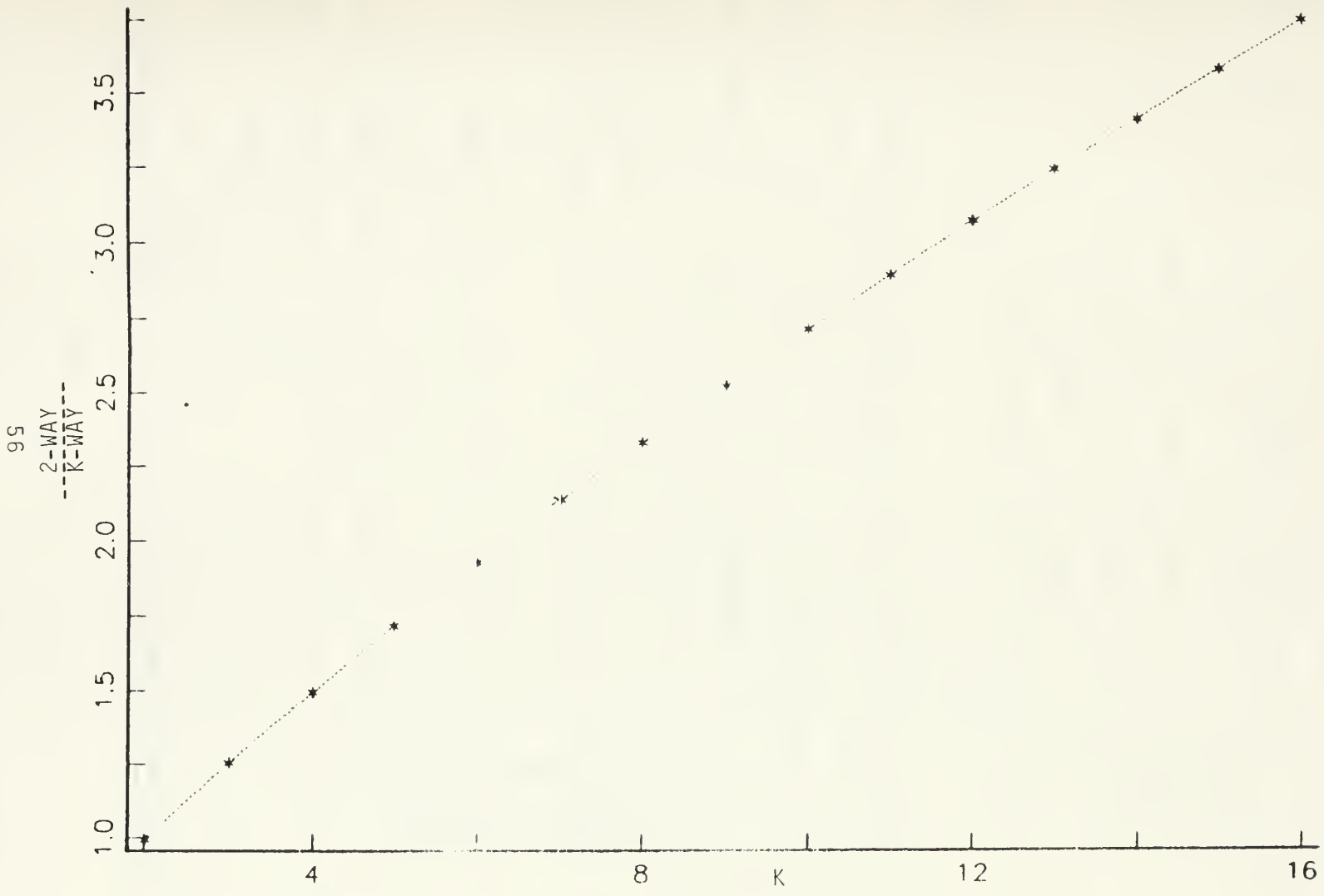

Figure 5.3. Computing Complexity for Merging with various K

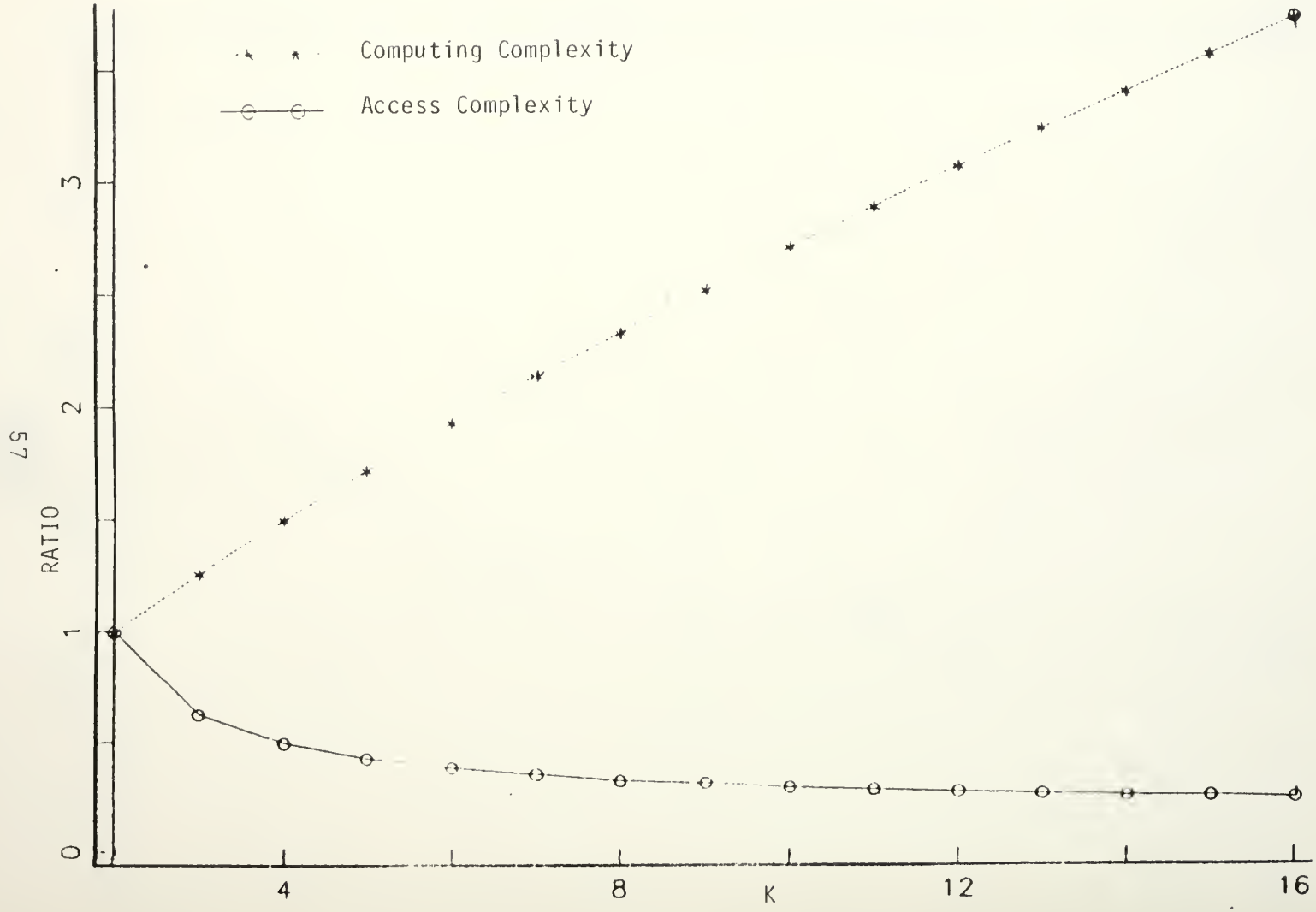

Figure 5.4. Computing and Access Complexities of Merging with various K

C. FITTING THE SOFTWARE ARCHITECTURE OF MDBS

Up to this point we have not considerd how the existing features of MDBS software architecture might be utilized for the sort function. We may ask a question such as whether the descriptor and cluster information can be used to improve sorting? Another question is whether existing I/O mechanisms can be used to support the temporary storage requirements. A third question is whether an alternative strategy should be adopted when the number of records are not evenly distributed across the backends. We will examine these three questions in detail.

1. Utilizing the Descriptor and Cluster Information

Recall that the database in MDBS is organized into clusters. Each cluster has a unique cluster id , and associated with a unique set of cluster ids. A record belongs to one and only one cluster. The cluster to which a record belongs is determined by the set of descriptor ids which can be derived from the directory keywords of the record.

How might this helps us in sorting? First consider the case that the primary(first-listed) attributes in the ordering specification are not directory attributes. In this case, the cluster to which a record belongs has no bearing on the final sorted order.

Next consider the case where the attributes in the ordering specification are all directory attributes. In this

case, we can use cluster information in the following manner. If we also know the relative order of the descriptor ids which determine the clusters, we may simply concatenate the records from the cluster having the lowest order descriptor ids with the cluster having the next higher order descriptor ids, and so on.

Finally, consider the case where the primary (first-listed) attributes in the ordering specification are directory attributes, and the secondary attributes in the ordering specification are non-directory attributes.

Let us first take a look at what we may need to utilize the existing machanisms. What is useful for sorting process is to know cluster ids and consequently the group of descripter ids(DIDs). The necessary point is to know the DIDs associated with records. If the record process is informed with the DIDs of records as well as their addresses and also the records are retrieved in terms of cluster numbers, that is, there is no record retrieved belongs to another cluster till all the records belonging to a cluster are retrieved. This process guarantees that if the records are going to be sorted with an attribute which is directory table attribute, and if the attribute is either type_A or type_B attribute then none of the clusters will have a record with the same attribute value. We also need another process to define which cluster has less or larger value of attributes. This process needs to check DIDs of clusters

from descriptor-to-descriptor-id table and gives a list of DIDs.

We may consider the utility of the above cases. First of all, we need to implement three different algorithms to handle these three different cases. Second, probabilty that primary sort specification attributes are directory attributes is unknown.

Let us assume that the system will be augmented with the implementation of the cluster information. In that case, modifications to MDBS are to be done. Recall that the record processing knows only the addresses of the records to be retrieved. Therefore, record processing is to be informed with not only cluster info but also descriptor information from directory management, including relative ordering of clusters based on descriptor ids. We do not have a mechanism available to support the idea. On the other hand, this implementation violates the information-hiding principles upon which directory management is designed.

2. Utilizing Existing Mechanism for Storing Temporary Data

In the previous sections we have assumed that the system was providing the temporary storage requirements for the sort function. We have not considered about how this might be accomplished. We know that system allocates tracks as required for new clusters or for extending existing clusters. Therefore, we know that there exists a mechanism

for allocating storage. The difficulty lies in that the allocation is related to the concept of a cluster, and not to a "block" of data.

In order to use the existing mechanisms, then, we must establish some relationships between blocks of sorted data and clusters. Since we are sorting block-at-a-time, we initially need to establish as many temporary clusters as we have blocks of data. Then, with each successive pass of the merge algorithm, we will require only half the previous number of clusters, although the total space required remains the same.

In current MDBS, storage is allocated only in the case of an insert request, where the records is to be inserted into an already-full cluster or a new cluster is to be established. The list of available (free) secondary storage addresses is maintained by directory management. New addresses for new clusters are assigned during the address-generation phase.

The second consideration is that addresses are associated with specific clusters, and new cluster ids are assigned only by the controller. The third consideration is that records are inserted record-at-a-time, based on an insert request. For the sorting process, we wish to write blocks of records.

In order to use the existing mechanisms, we must modify MDBS so that

(1) The sort process can request a new temporary cluster. This may involve sending a message to the controller.

(2) Directory management can generate addresses as required for the temporary clusters.

(3) Record processing can insert blocks of records as well as single record.

(4) Temporary clusters and their storage can be freed when no longer needed.

This is a disadvantage due to extensive modifications.

As an alternative, we may consider the following case. Reserve a certain number of addresses as temporary storage at system setup time. Use these addresses and the low-level read and write functions of record processing for temporary storage.

3. The Case that Records are not Evenly Distributed Across the Backends

Our time complexity formulas reflect the perfect conditions for distribution of the records which are to be sorted. They do not give the correct results for the condition that one backend contains all the records to be sorted and the other backends do not contain any records. In such a case there are two alternatives to be considered. The

first is that the backend having the records performs the sort function without redistribution of records. The second is that the records are redistributed evenly among the backends. In the following sections we will examine these two alternatives in detail.

a. The Backend Performs the Sort Function

Assuming that the algorithm in Chapter IV section C.2 has been selected for implementing sort function in MDBS, we will calculate the time complexities for the sort function. The backend now contains (B*b) blocks. So, the internal sort phase time complexity is

$$O (B*b*r*\log r),$$

and 2*B*b accesses to the secondary memory are required which is

$$O (B*b).$$

The merge process requires the time $O(B*b*r*\lceil \log (B*b) \rceil)$ with the access time to the secondary memory $O(B*b*\log (B*b)\rceil)$.

Therefore, the sort function time complexity is

$$O (B*b*r*(\log r + \lceil \log (B*b) \rceil)),$$

and the required accesses to the secondary memory are

$$O (B*b*\lceil \log(B*b) \rceil).$$

b. The Records are Distributed Evenly Among the Other Backends

In this alternative the backends should inform the controller if thay do not have any records to be sorted. The controller then manages the transfer of the records from one backend to the other backends.

Since one backend contains (B*b) blocks, b(B-1) records are to be transmitted to the other backends. This requires communication time of O(B*b). The backends now contain equal number of blocks, b. The time complexities can now be calculated as in the Chapter IV section C.2.

The internal sort process time is O(b*r*log r). The merge process time at the backends is O(b*r*log b). Accesses required at the backends are O (b*(1+log b)).

Depending on the average time required to transmit a block from a backend to another backend, we can analyze the difference between the aforementioned alternatives. At this moment we do not know the value of the transmission time a block. Clearly, there are some cases in which the transmission is not cost-effective.

VI. INTRODUCTION TO THE JOIN

In this part of the thesis, we investigate possible ways of implementing the join operation in MDBS. We consider how the functions of the join operation can be distributed over the controller and the backends. Again, we wish to take all possible advantage of the parallelism inherent in the MDBS hardware and software architecture. We also wish to adhere to the design goals of MDBS, in particular the minimization of the controller function and message traffic.

In this chapter, we define the terminology and notation which we will use in our analysis, and make some simplifying assumptions. In Chapter VII, we consider alternative distributions of the functions of the join operation over the controller and the backends. We examine an alternative join algorithm, a sort-and-match algorithm, in Chapter VIII. Finally, a recommendation for implementation is given in Chapter IX.

A. TERMINOLOGY AND NOTATION

First, let us define some terminology. A join involves two relations, the source relation and the target relation. The join is formed over an attribute (or attributes) that belong both to the source relation and to the the target relation. We will call these the source attribute(s) and the

target attribute(s), respectively. The domains of the source attribute(s) must be the same as the domains of the target attribute(s).

There are many types of joins. First we examine the natural join. Let us use an example to illustrate the natural join. The relations participating in a natural join are given in Figure 6.1.a. Relation S, the source relation, consists of three-tuples of attributes, A, B, and C. Relation T, the target relation, consists of three-tuples of attributes, B, C, and D. The assumption is made that attributes having the same name are defined over the same domain of values. Thus, the attributes B and C in relation S are assumed to be drawn from the same domain of values as the attributes B and C in relation T. Figure 6.1 shows the cross product SxT of relations S and T, SxT. SxT is formed by concatenating each tuple of relation S with every tuple of relation T.

The natural join is formed in two steps. First select from SxT the tuples such that the values of both columns headed by B and both columns headed by C are the same. There are three such tuples, the first, fifth, and ninth shown in Figure 6.1.(b). The second step is to project from those tuples one column for each distinct attribute. The result relation, S|X|T, is shown in Figure 6.1.(c).

S

A	B	C
1	2	3
4	5	6
7	8	9

T

B	C	D
2	3	1
5	6	4
8	9	7

(a)

SxT

A	B	C	B	C	D
1	2	3	2	3	1
1	2	3	5	6	4
1	2	3	7	8	9
4	5	6	2	3	1
4	5	6	5	6	4
4	5	6	8	9	7
7	8	9	2	3	1
7	8	9	5	6	4
7	8	9	8	9	7

(b)

S|X|T

A	B	C	D
1	2	3	1
4	5	6	4
7	8	9	7

(c)

Figure 6.1. Natural join of two relations, S and T.

In general, a join operation can be specified using the arithmetic comparison operators, = ,~=, < , <= , > , >=. Any of these operators may be used to specify the relationships between the values of the source attribute(s) and the target attribute(s). For example, the natural join shown in the example above could be specified as the join of

S and T where attribute values of B and C in S are identical to the attribute values of B and C in T, i.e., S.B=T.B and S.C=T.C. When the equal comparison operator is used, the join operation is called an equality join. When any other comparison operator is used, the join operation is called an inequality join. The join operation is associative, so that more than two relations may be joined. For example the join of three relations S, T, and U, is the same as the join of S and T, and the join of U and the first join.

There are a variety of join algorithms. The simplest is the straightforward or nested-loops join. The algorithm is shown in Figure 6.2.

```
For each tuple in the source relation do
  For each tuple in the target relation do
      If the join condition holds true then

                    form a result tuple
```

Figure 6.2. Straightforward Join Algorithm

In the chapters which follow, we will simplify our analysis by assuming that join operations are restricted to equality joins over a single source attribute and a single target attribute. The terms, source relation and target

relation, refer to the files participating in a join operation in MDBS. Hence, the source file refers to a source relation, and the target file refers to a target relation. We will also adopt the following notation:

Cs : The number of records in a source file

Ct : The number of records in a target file

n : The number of blocks belonging to a source file at a backend, Cs/(B*r).

m : The number of blocks belonging to a target file at a backend, Ct/(B*r).

q : Quotion of the cross-product of á source file and a target file which participate in a join operation.

log : Logarithm to the base 2.

B. ASSUMPTIONS

In analyzing the alternatives of the distributions of the join function, we make the following assumptions.

1) The source and target records are distributed equally across the backends.

2) The join operation is an equality join over a single source attribute and a single target attribute.

3) The join function is performed after the retrieval and selection operations specified in the request have been performed.

4) The straightforward or nested-loops join algorithm is used to perform the join.

5) Accesses to the secondary storage are carried out block-by-block.

6) The source and target files do not contain any duplicate records (i.e., after retrieval of the records which are to participate in the join operation, there are no two identical records in the source file or in the target file). Therefore, there is no record elimination process from the files.

C. A SYNTAX FOR THE JOIN

In this section, we will give a syntax for a 2-way join. MDBS utilizes an attribute-based data language, ABDL, for user queries. Indeed, an ABDL can be used for any database applications as a kernel language of any kind of database machines. Current database application language queries, for instance, SQL, can be be mapped to ABDL requests.

Using ABDL, a 2-way equality join request is shown as the following.

```
RETRIEVE (attribute_list_1) (query_1)
CONNECT ON (attribute_1, attribute_2)
        (attribute_list_2) (query_2)
```

The RETRIEVE clause implies that the records whose attribute-value pairs given in attribute_list_1 satisfy the conditions given in query_1, and the records whose attribute-value pairs given in attribute_list_2 satisfy the conditions given in query_2, are extracted from the database. Let R1 and R2 be the two different files containing these records, respectively. The CONNECT-ON clause specifies the join on the relations R1 and R2 with the attributes attribute_1, which is (implicitly if not explicitly) in attribute_list_1, and attribute_2, which is in attribute_list_2.

VII. THE ALTERNATIVE DISTRIBUTIONS OF THE JOIN FUNCTION

In analyzing the alternative distributions of the join function, we will again consider three different possibilities.

A. The controller performs the join function

B. The backends perform the join function

C. The join function is shared by the controller and the backends.

We will examine each of these alternatives in detail at the following sections.

A. THE CONTROLLER PERFORMS THE JOIN FUNCTION

In this alternative, the backends perform the retrieval of the records which will participate in the join operation. These records are then sent to the controller, and the controller performs the join.

Since each backend contains n source file blocks and m target file blocks, the communication complexity is

$$O(B*(n+m)), \text{ or}$$

$$O((Cs+Ct)/r).$$

Then, storing these blocks in the secondary storage of the controller has

$$O((Cs+Ct)/r)$$

access complexity.

After receiving the records from all backends, the controller now has (B*n) source file blocks and (B*m) target file blocks. Using the straightforward join algorithm, each record in the source file is compared with each record in the target file in order to form the join. This requires (Cs*Ct) comparisons. So, the computing complexity is

$$O(Cs * Ct).$$

Assuming that no more than one block of the source file and one block of the target file are in the primary storage at one time, $2*(B^2*n*m)$ accesses to the secondary memory are required. In terms of the cardinalities of the source and target files, this is the access complexity of

$$O((Cs*Ct)/r^2).$$

B. THE BACKENDS PERFORM THE JOIN FUNCTION

In this alternative, we will consider three different strategies. In the first, the backends share the join operation equally. In the second, the join function is performed step-by-step at the backends. In the third, a single backend performs the join function with the complete source and target files. Let us examine the details of these strategies.

1. The Backends Share the Join Equally .

In this strategy, the backends send either source or target records to each other. Let us assume that the target records are transmitted between the backends. After transmission of the records, each backend contains Ct target records. Next, each backend performs the join function over its own part of source records and all of the target records. Then, the result records from the backends are transmitted to the controller.

Since each backend contains m target file blocks, (B*m) target file blocks are transmitted. Therefore, complexity of transmitting the target file blocks among the backends is

$$O(B*m), \text{ or}$$
$$O(Ct/r).$$

Each backend first stores (B*m) target file blocks which requires access complexity of

$$O(Ct/r).$$

Each backend now contains n source file blocks and (B*m) target file blocks. Therefore, the effective computing complexity for performing the join is

$$O(B*n*m*r^{2}), \text{ or}$$
$$O(Cs*Ct/B).$$

2*B*m*n accesses to the secondary storage are required, so the access complexity is

$$O(Cs*Ct/(B*r)).$$

Finally, each backend transmits the result records to the controller. Let us assume that each backend yields the same number of result records, expressed as a percentage q of the cross-production of the records participating in the join. Then, the number of the records to be transmitted from each backend to the controller will be (q*B*n*m*r) or (q*(Cs*Ct)/B). The communication complexity for transmission the result records from B backends to the controller, then, is

$$O(q*(Cs*Ct)/r).$$

2. The Backends Perform the Join Step-by-Step

In this strategy, the join operation is performed step-by-step at the backends. At each step, the number of backends involved in the join is reduced by one-half. A backend performing the join function sends its source and target records to its neighbor backend. Figure 7.1 depicts the the flow of records. The total number of steps required is log B, where B is the number of backends.

The arrows indicate the transmission direction of blocks. At each step, the backends involved first perform the join on the portions of the source and target files available, and send the partial result to the controller.

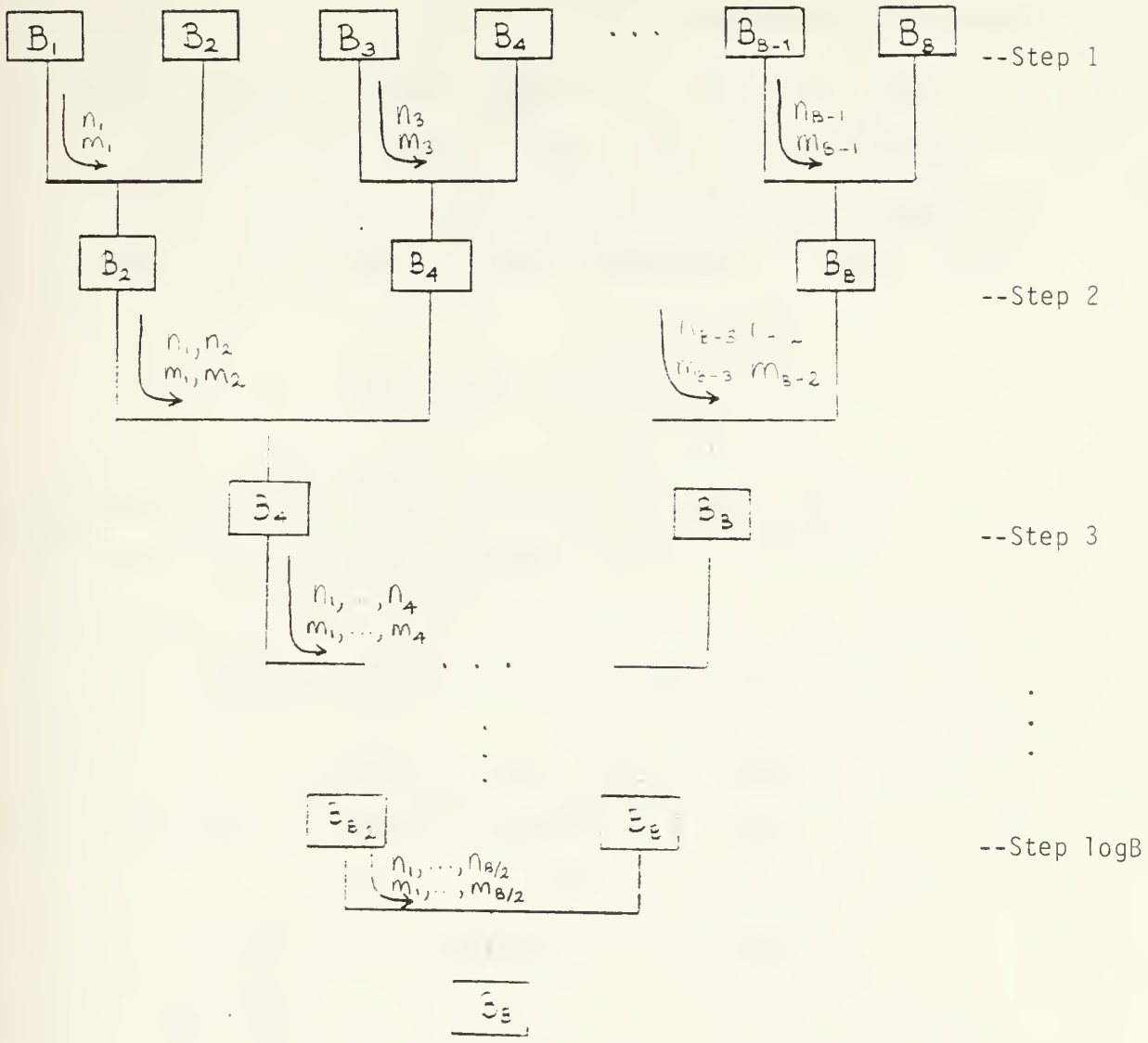

Figure 7.1. Performing the Join Function Step-by-Step at the Backends

Next, the subsets of the source and target files are sent to the neighbor backend.

At each step, the number of blocks to be transmitted over the broadcast bus is half of the total number of source file blocks plus half of the total number of target file blocks. Thus, the communication complexity for log B steps is

$$O(((Cs+Ct)/r)* \lceil \log B \rceil).$$

At each step, the backends receiving the source and target records from their neighbors first store them before the join starts. The effective access complexity of storing the records at each step is derived as follows.

1. step	$(1/2)*(Cs+Ct)/(B*r)$
2. step	$1 *(Cs+Ct)/(B*r)$
3.step	$2 *(Cs+Ct)/(B*r)$
4. step	$4 *(Cs+Ct)/(B*r)$
.	.
.	.
.	.
$\lceil \log B \rceil$ step	$2^{\lceil \log B-2 \rceil} *(Cs+Ct)/(B*r)$

Therefore, the total effective access complexity for storing the records is

$$O(2^{\lceil \log B \rceil} *(Cs+Ct)/(B*r)).$$

The computing complexity for the join is derived as following.

1. step (n*r)*(m*r) = $n*m*r^2$

2. step (2*n*r)*(2*m*r) = $4*n*m*r^2$

3. step (4*n*r)*(4*m*r) = $16*n*m*r^2$

. . .

. . .

. . .

$\lceil \log B \rceil$ step $(2^{\lceil \log B-1 \rceil}*n*r)*(2^{\lceil \log B-1 \rceil}*m*r)$ = $2^{\lceil 2\log B-2 \rceil}*n*m*r^2$

Therefore, the total effective computing complexity is

$$O(2^{\lceil 2\log B \rceil}*Cs*Ct/B^2).$$

Since the number of source and target blocks participating in the join changes at each step, the access complexity during the join is derived as following.

1. step 2*(n*m)

2. step 2*(2n*4m)

3. step 2*(4n*4m)

4. step 2*(8n*8m)

. .

. .

. .

$\lceil \log B \rceil$ step $2*(2^{\lceil \log B-1 \rceil}n*2^{\lceil \log B-1 \rceil}m)$

The total effective access complexity is, then,

$$O(2^{\lceil 2\log B \rceil} * (Cs*Ct)/(B^2*r^2)).$$

Only the result records are transmitted to the controller. Since we use q*Cs*Ct to represent the number of result records, communication complexity is

$$O(q*(Cs*Ct)/r).$$

3. One Backend Performs the Join Function

In this strategy, the source and target records at each backend are transmitted to a designated backend, which this performs the join. Since each backend contains n source file blocks and m target file blocks, the communication complexity is

$$O(B*(n+m)), \text{ or}$$
$$O((Cs+Ct)/r).$$

The records sent from the other backends are first stored into the secondary storage of the designated backend. This is the access complexity of

$$O((Cs+Ct)/r).$$

The designated backend now contains Cs source records and Ct target records. Using a straightforward join algorithm, the computing complexity is

$$O(Cs*Ct),$$

and $2*B^2*n*m$ accesses to the secondary storage are required, for access complexity of

$$O(Cs*Ct/r^2).$$

The designated backend produces q*Cs*Ct result records. Transmission of these result records to the controller has complexity of

$$O(q*(Cs*Ct)/r).$$

C. THE CONTROLLER AND THE BACKENDS SHARE THE JOIN FUNCTION

In this alternative, the controller and the backends share the join function , and the controller integrates the results. Each backend transmits the its part of both the source records and the target records to the controller. At the same time, each backend performs a partial join with its source and target records. In the meantime, the controller performs the join function with the sets sent from the backends, except for those sets which are joined at the backends.

Let n1,n2,...,nB be the subsets of the source file and m1,m2,...,mB be the subsets of the target file such that backend i, Bi, contains the subsets ni and mi. The transmission of the whole source and target file into the controller has the communication complexity of

$$O(B*(n+m)), \text{ or}$$
$$O((Cs+Ct)/r).$$

The controller first stores the records. This requires the access complexity of $O((Cs+Ct)/r)$.

The partial join function at the backend has the computing complexity of $O(Cs*Ct/B^2)$, and access complexity of $O(Cs*Ct/(B^2*r^2))$.

The controller now contains ni source set and mi target set. Since the backends perform only part of the join , the rest of the join function is performed at the controller. This means each ni is compared with mj to output the result records such that $1<= i <=B$ and $1<= j <= B$, and $i~=j$. This requires $B*(B-1)$ times $(n*m)$ comparisons. Therefore, the join function at the controller has computing complexity of

$$O(n*m*B*(B-1)*r^2), \text{ or}$$
$$O(Cs*Ct),$$

and access complexity of $O(B^2*n*m)$, or $O(Cs*Ct/r^2)$.

D. EVALUATING THE ALTERNATIVE DISTRIBUTIONS OF THE FUNCTIONALITY

In the previous sections we have presented five alternative distributions of the functionality of join between the controller and the backends. In this section, we will analyze the tradeoffs of the alternatives. Table 3 summarizes the results of the analyses in terms of that the computing, access, and communication complexities.

Alternative A represents the distribution of function presented in Section A of this chapter. The controller performs the join function. Alternative B.1 represents the distribution presented in Section B.1 of this chapter. The backends share the join function equally. Alternative B.2 represents the distribution presented in Section B.2 of this chapter. The backends perform the join function step-by-step. Alternative B.3 represents the distribution B.3 presented in Section B.3 of this chapter. Finally, alternative C represent the distribution C presented in Section C of this chapter. The controller and the backends share the join function. Let us examine each of these alternatives with regard to the design goals of MDBS.

Alternative A is clearly contrary to design goal of minimizing controller function. Therefore, we will eliminate it from further consideration. Alternative B.1 meets the goal of minimizing controller function and distributing the work over the backends. The communication complexity is also

less than that of either of the other alternatives, B.2 and B.3.

Alternative B.2 meets the design goal of minimizing controller function. However, the computing and access complexities increase exponentially with the factor of 2*log B. This is an especially important consideration for I/O overhead in the system. In addition, the same blocks will be broadcasted log B times over broadcast bus, increasing high communication overhead. As we recall, a similar procedure was proposed in Chapter IV for the sort function. However, the characteristic of the join function does not take advantage of this procedure. At each step, the output of the backends is wasted, since each record in the source file must be compared with every record in the target file to form the join. The same records will be transmitted between the backends redundantly. Therefore, we will eliminate this alternative from further consideration. Alternative B.3 does not meet the design goal of sharing the work between the backends. Furthermore, transmission of source and target file blocks into the designated backend increases the communication overhead. This alternative is also eliminated from further consideration.

Alternative C increases the amount of work which is to be done by the controller. This is also contrary to design goal of minimizing controller function. Therefore, we will eliminate this alternative from further consideration.

As easily seen from the above explanations, alternative B.1 is the best approach to the distibution of functionality. As we recall, a straightforward join algorithm is utilized to analyze the alternative distributions. Having decided the best alternative for distributing the join function with the simplest join algorithm, we can improve the efficiency of the chosen alternative by using a different algorithm, the sort-match algorithm. This situation will be examined in the following chapter.

ALTERNATIVE	COMPUTING COMPLEXITY		ACCESS COMPLEXITY		COMMUNICATION COMPLEXITY	
	AT THE BACKENDS	AT THE CONTROLLER	AT THE BACKENDS	AT THE CONTROLLER	AMONG THE BACKENDS	AMONG THE CONTROLLER AND THE BACKENDS
A	---	$O\left(Cs * Ct\right)$	---	$O\left(\frac{Cs * Ct}{r^2}\right)$	---	$O\left(\frac{Cs + Ct}{r}\right)$
B.1	$O\left(\frac{Cs * Ct}{B}\right)$	---	$O\left(\frac{Cs * Ct}{B * r} + \frac{Ct}{B}\right)$	---	$O\left(\frac{Ct}{r}\right)$	$O\left(q * \frac{Cs * Ct}{r}\right)$
B.2	$O\left(2^{2\log B} * \frac{Cs * Ct}{B^2}\right)$	---	$O\left(2^{2\log B} * \frac{Cs * Ct}{B^2 * r^2}\right)$	---	$O\left(\frac{Cs * Ct}{r} \log B\right)$	$O\left(q * \frac{Cs * Ct}{r}\right)$
B.3	$O\left(Cs * Ct\right)$	---	$O\left(\frac{Cs * Ct}{r^2}\right)$	---	$O\left(\frac{Cs + Ct}{r}\right)$	$O\left(q * \frac{Cs * Ct}{r}\right)$
C	$O\left(\frac{Cs * Ct}{B^2}\right)$	$O\left(\frac{Cs * Ct}{r^2}\right)$	$O\left(\frac{Cs * Ct}{B^2 * r^2}\right)$	$O\left(\frac{Cs * Ct}{r^2}\right)$	---	$O\left(\frac{Cs + Ct}{r}\right)$

TABLE 3. THE TIME COMPLEXITIES FOR THE ALTERNATIVE DISTRIBUTIONS OF THE JOIN FUNCTIONALITY

VIII. AN ALTERNATIVE JOIN ALGORITHM

In the previous chapter, we analyzed the distribution of the functions of the join operation in MDBS assuming that a straightforward join algorithm is used. In the first part of the thesis, we discussed how the sort function can be implemented in MDBS. Assuming that the sort function is implemented as recommended in chapter IV, we now discuss how the join operation can be implemented using a sort-match algorithm.

A. ALTERNATIVE DISTRIBUTIONS OF THE JOIN FUNCTION BY USING A SORT-MATCH ALGORITHM

When using a sort-match algorithm, the source records and the target records are first sorted. Then, the join function is performed. The join can be formed by a simple matching of the source attribute values and the target attribute values.

In Chapter IV, we examined how to perform the sort function at MDBS. As we recall, our proposal was to apply the alternative C.2 in Chapter IV, the backends sort and perform a partial merge, and the controller performs the final merge. With such a capability, we propose two alternatives for distributing the functions of the sort-match join algorithm among the controller and the backends.

The first alternative is as follows. Each backend performs sort and partial merge of the source and target records. Then, each backend broadcasts its target records to all other backends. Each backend then joins its portion of the source records with all of the target records, transmitting the results to the controller.

The second alternative is the following. The backends perform sort and partial merge on the source and target records, which are then transmitted to the controller. The controller performs the final merge of the source records and of the target records, and then performs the join of all of the source records and all of the target records. Let us examine each of these alternatives in detail.

1. The Backends Share the Join

In this case, both source and target files are first sorted at the backends separately. Using a comparison-based sorting algorithm, the effective computing complexity of the internal sort phases of both Cs/B source and Ct/B target records is

$$O(((Cs+Ct)/B) * \log r).$$

2*(n+m) accesses to the secondary storage are required. So, the effective access complexity during the internal sort phases of both source and target files is

$$O((Cs+Ct)/(B*r)).$$

Assuming that the merge phase is also performed to complete the sorting of both files, the effective computing complexity of the merge phase at the backends is

$$O(n*r*\lceil \log n \rceil + m*r*\lceil \log m \rceil), \text{ or}$$
$$O((Cs/B)*\lceil \log(Cs/(B*r)) \rceil + (Ct/B)*\lceil \log(Ct/(B*r)) \rceil)$$

$2*(n*\log n + m*\log m)$ accesses to the secondary storage are required to complete the merge function. Therefore, the effective access complexity for the merge at the backends is

$$O(n*\lceil \log n \rceil + m*\lceil \log m \rceil), \text{ or}$$
$$O((Cs/(B*r))*\lceil \log(Cs/(B*r)) \rceil + (Ct/(B*r))*\lceil \log(Ct/(B*r)) \rceil).$$

Next, the target records are transmitted between the backends . This is the communication complexity of

$$O(B*m), \text{ or}$$
$$O(Ct/r).$$

The target records transmitted from the other backends are first stored before the join starts. This is the access complexity of

$$O(Ct/r).$$

Each backend now contains n blocks of the source and B*m blocks of the target file. That is, each backend has one run of source file and B runs of target file blocks with the length of n and m, respectively. B*m target blocks, then,

must be merged by each backend. Assuming that a 2-way merge is used, the computing complexity of merging B*m blocks at a backend is

$$O(B*m*r*\lceil \log B \rceil), \text{ or}$$
$$O(Ct*\lceil \log B \rceil).$$

2*B*m*log B accesses to the secondary storage are required. So, the access complexity required during the merge of target records is

$$O((Ct/r) * \lceil \log B \rceil).$$

Finally, each backend performs the join over Cs/B source and Ct target records. The effective computing complexity of the join is

$$O(\min (n*r, B*m*r)), \text{ or}$$
$$O(\min (Cs/B , Ct)),$$

and 2*(max (n, B*m)) accesses to the secondary storage are required. This is the access complexity of

$$O(\max (Cs/(B*r), Ct/r)).$$

Each backend now has a portion of the result records. Using the same notations as in the previous chapter, there are q*(min (Cs/B , Ct)) result records at each backend. The communication complexity of transmitting the result records from each backend to the controller is

$$O((B/r)*q*\min (Cs/B , Ct)).$$

2. The Controller Performs the Join

Here, each backend performs the internal sort phase and the partial merge phase of the its portion of the source and the target records, and then transmits these records to the controller. The controller first merges the source and target records separately, and then performs the join on the source and the target records.

The effective computing complexity to sort n source file blocks and m target relation blocks at the backend is

$$O(n*r*\log r + m*r*\log r), \text{ or}$$
$$O(((Cs+Ct)/B)*\log r)).$$

2*(n+m) accesses to the secondary storage are required. So, the effective access complexity is

$$O((Cs+Ct)/(B*r)).$$

Assuming that a 2-way merge is implemented to complete the sort of n source and m target file blocks. So the computing complexity of the merge is

$O(n*r*\lceil \log n \rceil + m*r*\lceil \log m \rceil)$, or

$O((Cs/B)*\lceil \log(Cs/(B*r)) \rceil + (Ct/B)*\lceil \log(Ct/(B*r)) \rceil)$.

$2*(n*\log n + m*\log m)$ accesses to the secondary storage are required. This is the access complexity of

$O(n*\lceil \log n \rceil + m*\lceil \log m \rceil)$, or

$O((Cs/(B*r)*\lceil \log(Cs/(B*r)) \rceil + (Ct/(B*r)*\lceil \log(Ct/(B*r)) \rceil)$.

Next, the sorted records are transmitted to the controller. So, the communication complexity is

$O(B*(n+m))$, or

$O((Cs+Ct)/r)$.

The records are first stored at the controller before the join starts. This is the access complexity of

$O((Cs + Ct)/r)$.

The controller now contains $B*n$ blocks of source file and $B*m$ blocks of target file. That is, B runs of source file and B runs of target file with the length of n and m, respectively. The computing complexity of merging source and target records saperately is

$O(B*n*r*\lceil \log B \rceil + B*m*r*\lceil \log B \rceil)$, or

$O((Cs+ Ct)*\lceil \log B \rceil)$.

$2*B*(n+m)*\log B$ accesses to the secondary storage are

required. This is an access complexity of the merge at the controller which is O(((Cs+Ct)/r)*⌈log B⌉).

Finally, the controller performs the join on sorted source and target files. The computing complexity for the join is O(min (B*n*r, B*m*r)), or O(min (Cs, Ct)), and 2*(max (B*n, B*m)) accesses to the secondary storage are required. This is an access complexity of the join at the controller which is O(max (Cs/r, Ct/r)).

B. COMPARISONS BETWEEN THE TWO ALTERNATIVES

Table 4 illustrates the time complexities for both alternatives, using a sort-match algorithm. Again, the computing complexity, the access complexity, and the communication complexity are given separately. The computing complexity includes the sum of the computing complexities of the internal sort phase, the merge phase, and the join.

The access complexity includes the sum of the access complexities of the internal sort phase, the merge phase, and the join. Finally, the communication complexity shows the time required to transmit the source and the target records among the backends and between the controller and the backends. The complexity formulas of accesses to the secondary storage are given only for the additional accesses necessary to complete the join. In other words, accesses to the secondary storage to retrieve the records to perform

selection and projection before the join starts are not included. Let us examine the Table 4 row by row comparing the two alternatives.

The computing complexity in the backends for the alternative A.1 is larger than the alternative A.2 since each backend in alternative A.1 contains all the target file records. On the contrary, the alternative A.2 has a computing complexity at the controller. Therefore, alternative A.1 is better than alternative A.2 with regard to meeting the design goal of minimizing controller function.

The alternative A.1 requires more accesses to the secondary storage for the backends than the alternative A.2. However, again, the alternative A.2 requires more accesses to the secondary storage at the controller. Therefore, alternative A.1 is better than alternative A.2 due to meeting the design goal of minimizing controller function.

Despite the situation that the alternative A.2 has lower transmission overhead, this may be negligible when balanced against I/O requirements at the controller. Therefore, we will recommend the alternative A.2, i.e.,the backends perform the join, for implementation of the join using a sort-match algorithm in MDBS. This alternative best meets the design goals of minimizing controller function and sharing the work equally at the backends.

		A.1. THE BACKENDS SHARE THE JOIN BY USING A SORT-MATCH ALGORITHM	A.2. THE CONTROLLER PERFORMS THE JOIN BY USING A SORT-MATCH ALGORITHM
COMPUTING COMPLEXITY	AT THE Bs	$O\left(\frac{Cs+Ct}{B}\log r + \frac{Cs}{B}\log\frac{Cs}{B*r} + \frac{Ct}{B}\log\frac{Ct}{B*r} + Ct\log B + \min\left(\frac{Cs}{B}, Ct\right)\right)$	$O\left(\frac{Cs+Ct}{B}\log r + \frac{Cs}{B}\log\frac{Cs}{B*r} + \frac{Ct}{B}\log\frac{Ct}{B*r}\right)$
	AT THE C	— — — —	$O\left((Cs+Ct)\log B + \min(Cs, Ct)\right)$
ACCESS COMPLEXITY	AT THE Bs	$O\left(\frac{Cs+Ct}{B}\log r + \frac{Cs}{B}\log\frac{Cs}{B*r} + \frac{Ct}{B}\log\frac{Ct}{B*r} + \frac{Ct}{r} + \frac{Ct}{r}\log B + \max\left(\frac{Cs}{B*r}, \frac{Ct}{B}\right)\right)$	$O\left(\frac{Cs+Ct}{B*r} + \frac{Cs}{B*r}\log\frac{Cs}{B*r} + \frac{Ct}{B*r}\log\frac{Ct}{B*r}\right)$
	AT THE C	— — — —	$O\left(\frac{Cs+Ct}{r} + \frac{Cs+Ct}{r}\log B + \max\left(\frac{Cs}{r}, \frac{Ct}{r}\right)\right)$
COMMUNICATION COMPLEXITY	AMONG THE Bs	$O\left(\frac{Ct}{r}\right)$	— — — —
	AMONG THE C & THE Bs	$O\left(B*r*\frac{1}{r}*\min\left(\frac{Cs}{B}, Ct\right)\right)$	$O\left(\frac{Cs+Ct}{r}\right)$

Bs: Backends, C: Controller

Table 4. Alternative a.1. vs Alternative A.2.

		STRAIGHTFORWARD JOIN ALGORITHM	SORT-MATCH JOIN ALGORITHM
COMPUTING COMPLEXITY	AT THE Bs	$O\left(\frac{Cs * Ct}{B}\right)$	$O\left(\frac{Cs+Ct}{B}\log r + \frac{Cs}{B}\log\frac{Cs}{B*r} + \frac{Ct}{B}\log\frac{Ct}{B*r} + Ct*\log B + \min\left(\frac{Cs}{B}, Ct\right)\right)$
	AT THE C	——	——
ACCESS COMPLEXITY	AT THE Bs	$O\left(\frac{Cs*Ct}{B*r^2} + \frac{Ct}{r}\right)$	$O\left(\frac{Cs+Ct}{B*r} + \frac{Cs}{B*r}\log\frac{Cs}{B*r} + \frac{Ct}{B*r}\log\frac{Ct}{B*r} + \frac{Ct}{r}\log B + \max\left(\frac{Cs}{B*r}, \frac{Ct}{r}\right)\right)$
	AT THE C	——	——
COMMUNICATION COMPLEXITY	AMONG THE Bs	$O\left(\frac{Ct}{r}\right)$	$O\left(\frac{Ct}{r}\right)$
	AMONG THE C & THE Bs	$O\left(q * \frac{Cs*Ct}{r}\right)$	$O\left(B*q*\frac{1}{r} + \min\left(\frac{Cs}{B}, Ct\right)\right)$

Bs: Backends, C: Controller

Table 5. The Time Complexities of the Two Join Algorithms Performing the Join at the Backends.

C. COMPARISONS BETWEEN THE STRAIGHTFORWARD AND THE SORT-MATCH JOIN ALGORITHMS

Table 5 depicts the time complexities for the best alternative using the straightforward join algorithm and the best alternative using the sort-match join algorithm. Let us now compare the two alternatives. Let us assume that the number of source records is equal to the number of target records, i.e., Cs = Ct. Let the block size, r, be equal to 64. We will compare the access complexities and the computing complexities of the two alternatives with selected number of records involved, Cs and Ct, the result proportionality, q, and varying the number of backends, B.

Figure 8.1 shows the access complexities for $Cs=Ct=2^{14}$, q=0.1, and number of backends, B, from 2 to 16. The increasing number of backends has little effect on access complexity when a sort-match algorithm is used. However, when the straightforward algorithm is used, the access complexity decreases sharply as the number of backends increases. Note that for a large number of backends, B>16, the reduction becomes negligible. The access complexity required for the sort-match algorithm is always less than that required for the straightforward algorithm, and is substantially less for a smaller number of backends.

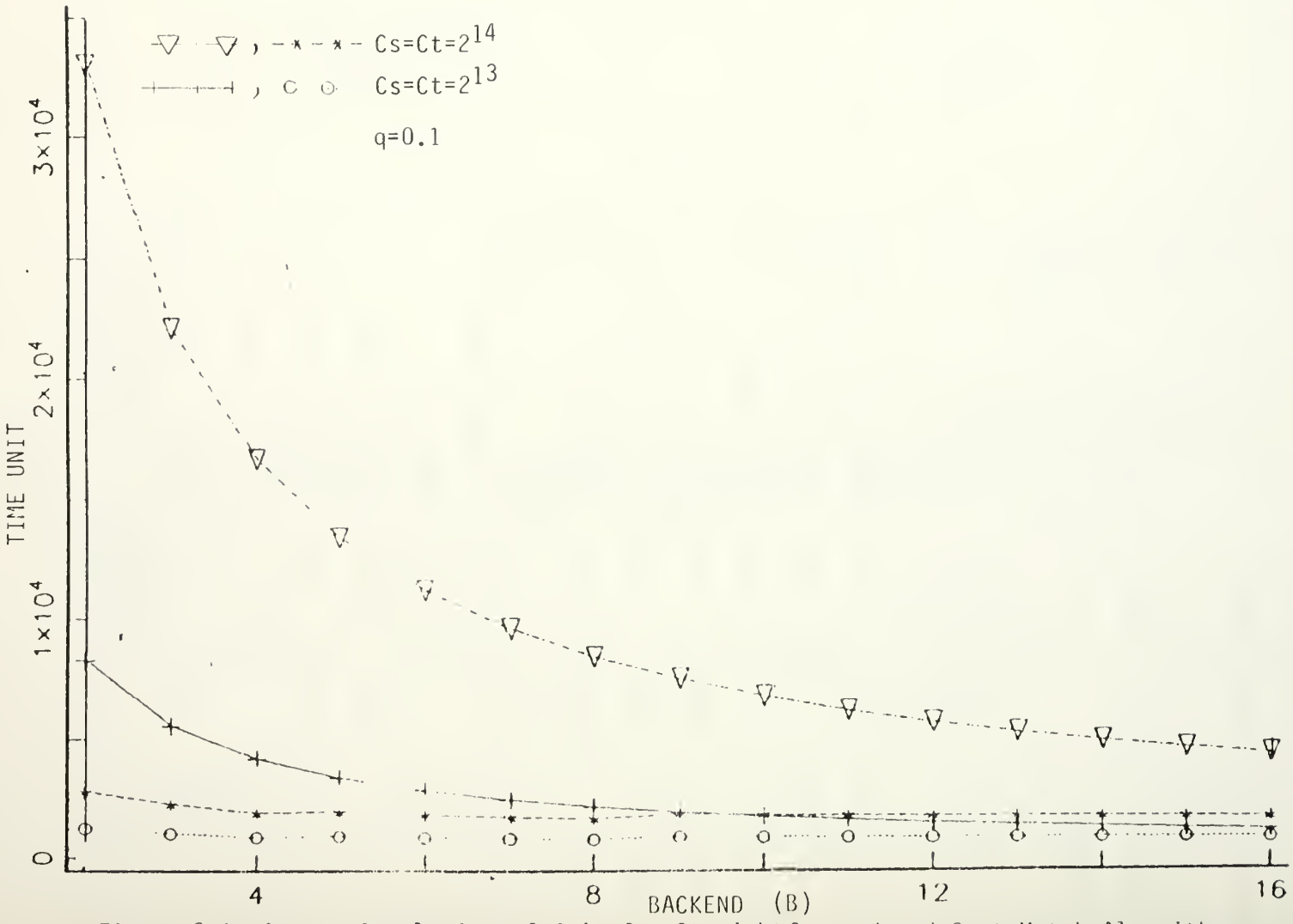

Figure 8.1. Access Complexity of Join for Straightforward and Sort-Match Algorithms

Figure 8.2 shows the computing complexity for both algorithms with $Cs=Ct=2^{13}$ and 2^{14}, q=0.1. In this case, both alternatives have decreasing computing complexity. Again, the computing complexity for the sort-match algorithm is less than that required for the straightforward join algorithm, and substantially less for a small number of backends. When the number of source and target records increase, the difference between the two algorithms also increase.

Figure 8.3 shows the communication complexity of the straightforward join algorithm with $Cs=Ct=2^{13}$, 2^{14}, and 2. The quotion, q, ranges from 0.1 to 0.5. Figure 8.4 depicts the same complexity for the sort-match join algorithm with $Cs=Ct=2^{13} - 2^{18}$, and q=0.1-0.5. These two figures illustrate that increasing Cs, Ct, and q affect on the communication complexity of the straightforward join algorithm more than the sort-match join algorithm.

D. RECOMMENDED PROPOSAL FOR THE DISTRIBUTION OF THE JOIN OPERATION

In the previous sections, we have analyzed the alternatives of the distribution of the functionality and shown the tradeoffs and the advantages of each one by using two different join algorithms, namely the straightforward join algorithm and the sort-match join algorithm.

Briefly, alternative B.1 in Chapter VII using a straightforward join algorithm and the alternative A.1 in Chapter VIII using a sort-match join algorithm are the best alternatives for distribution of the functionality. In both alternatives, the functional unit performing the join in MDBS is the backends. Finally, comparisons between these two alternatives have shown that the alternative A.1, join at the backends using a sort-match join algorithm, is better than the alternative B.1, join at the backends using a straightforward join algorithm, on account of meeting the design goal of minimizing the communication overhead between the controller and the backends.

Having analyzed all the alternatives, the most appropriate choice for implementing the join in MDBS is that each backend performs a partial join with its portion of source records and all target records. Then,the results are sent to the controller. The controller will then forward the final result to the host computer.

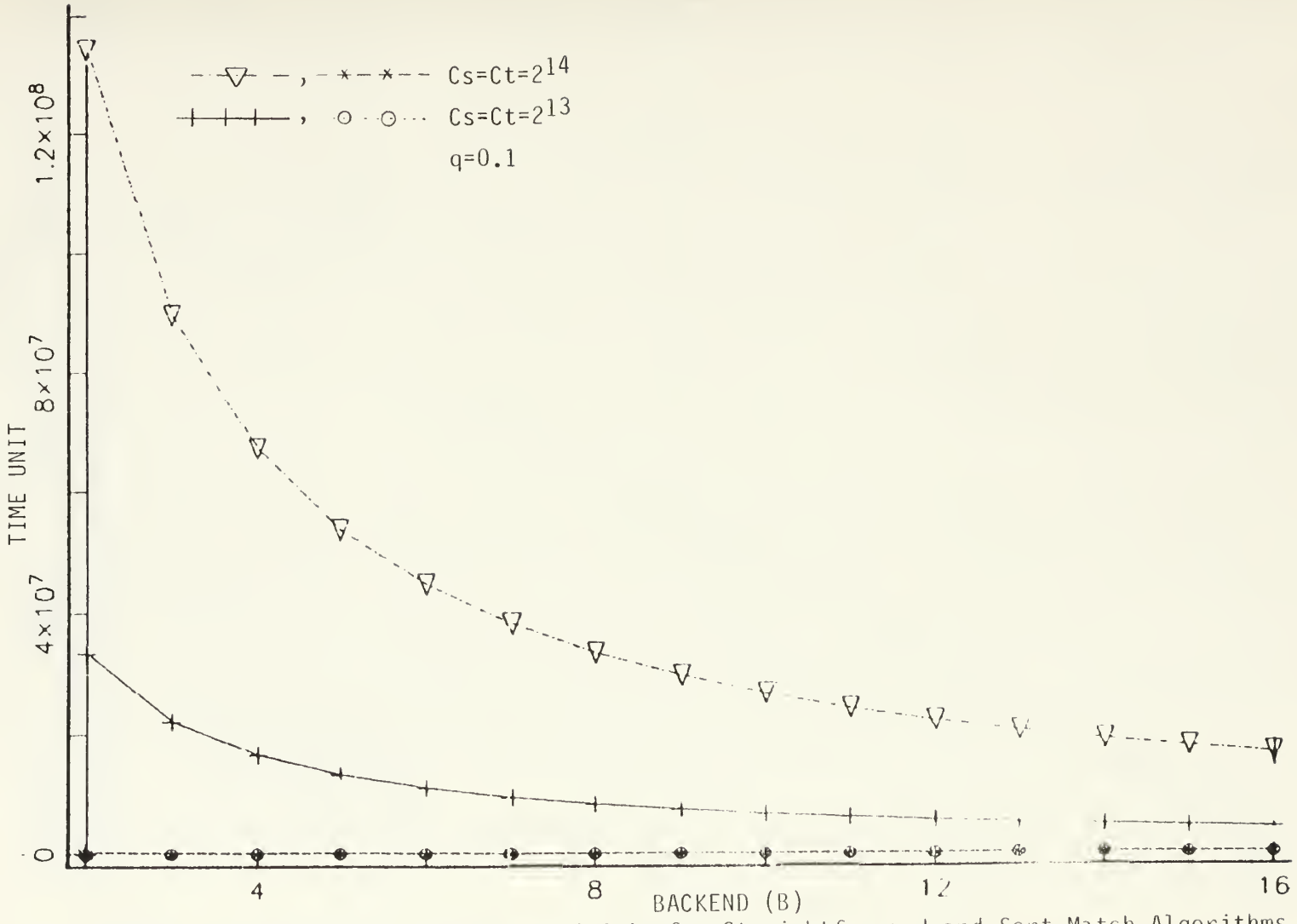

Figure 8.2. Computing Complexity of Join for Straightforward and Sort-Match Algorithms

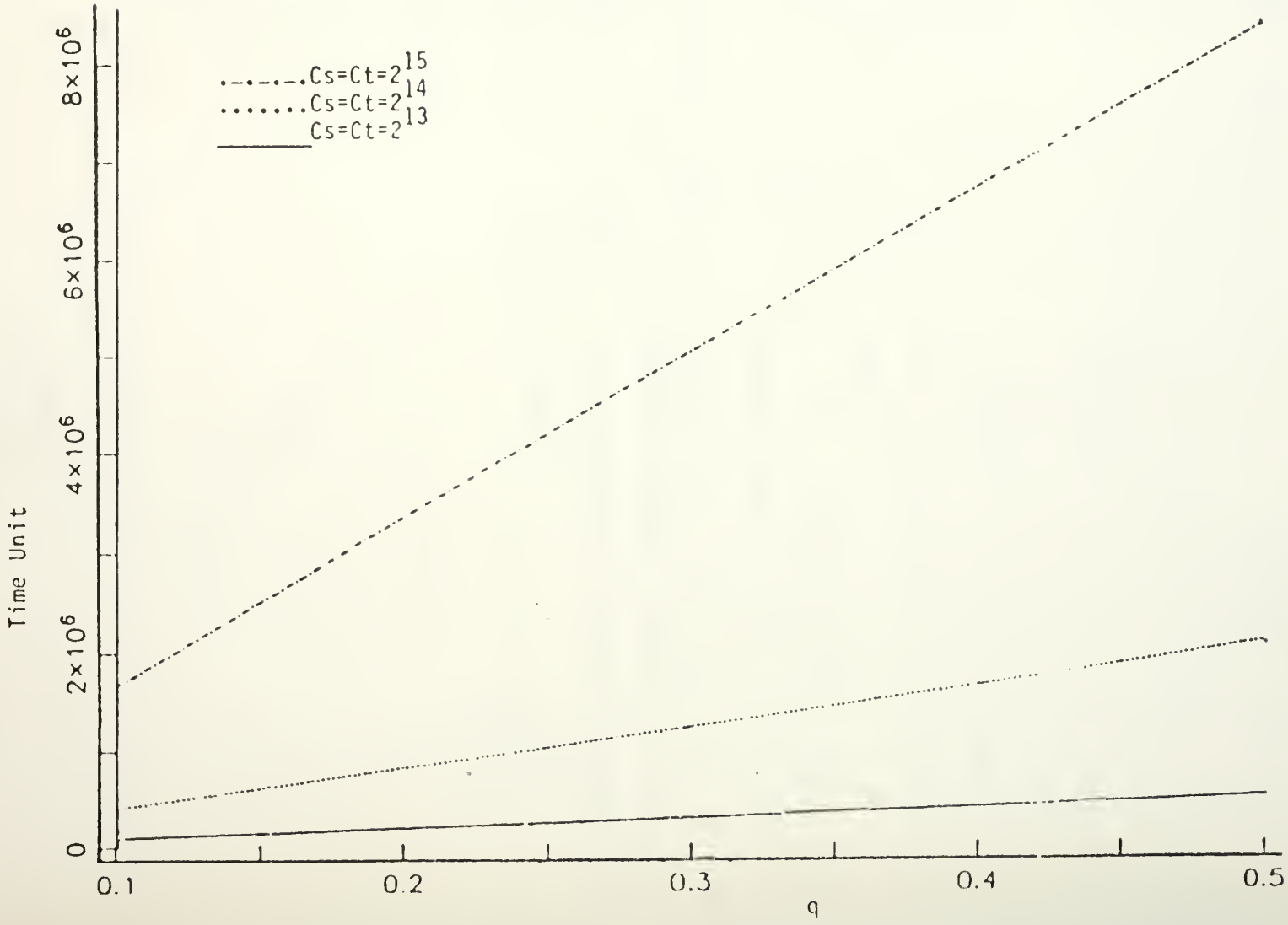

Figure 8.3. Communication Complexity for Join Using the Straightforward Algorithm

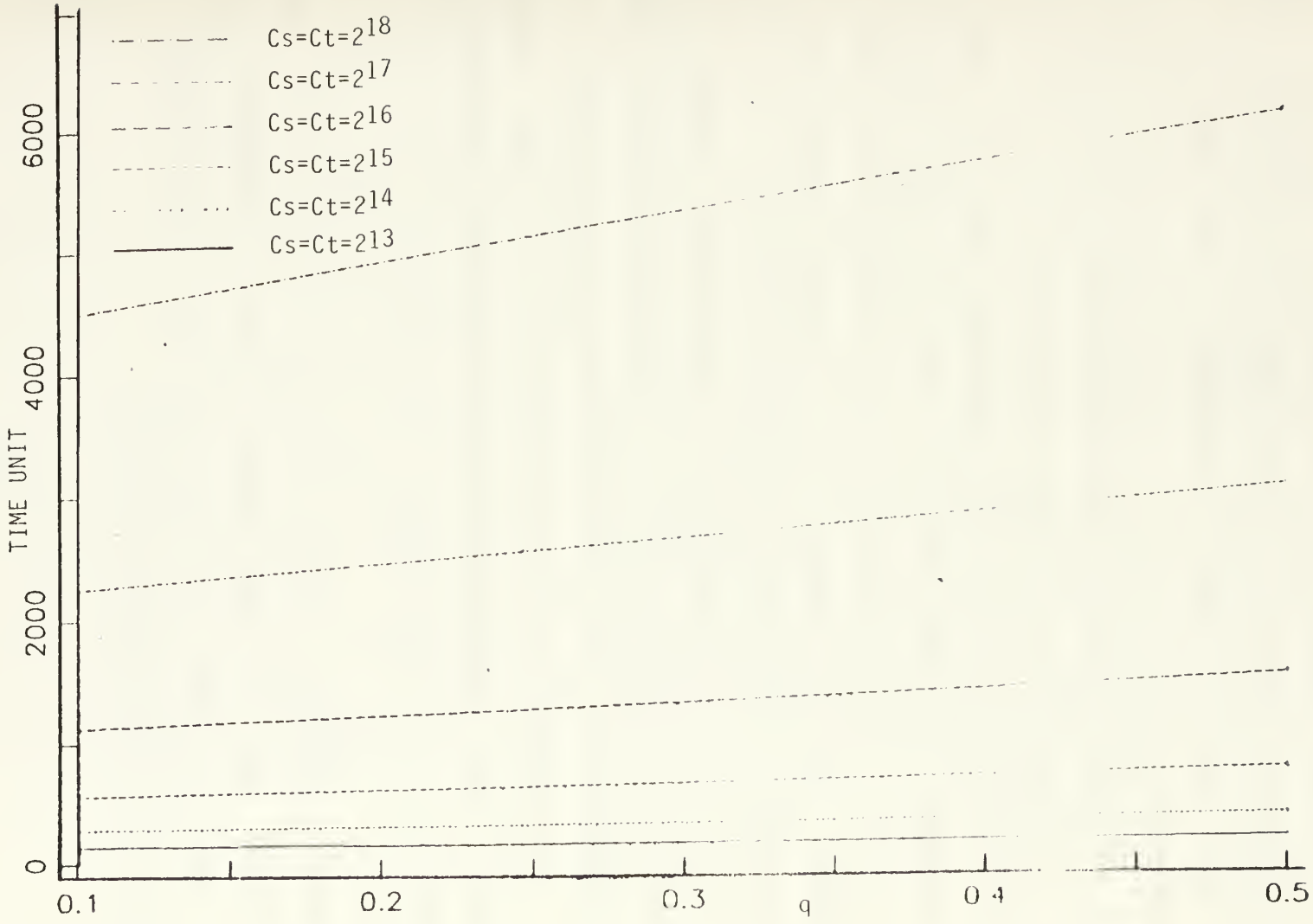

Figure 8.4. Communication Complexity for Join Using the Sort-Match Algorithm

IX. CONCLUSION

In this thesis, we introduce the sort and join operations into the Multi-Backend Database System (MDBS). Adding sort and join capabilities will increase the effectiveness of the system in supporting relational database and relational language interfaces. The key issue for alternatives is the way in which the functionality of the operation is distributed among the controller and the backends. We have observed that, in each case, that assigning the most of the work to the backends is always the better approach. Since the work is shared equally by the backends, increasing the number of backends in the system reduces the response time and increases the throughput, thus meeting the design goals. The selected solutions may also be implemented with less impact on the existing software.

Our proposal for the sort function is that the backends perform the sorting and partial merge, and the controller performs the final merge. Our proposal for the join function using the sort-match join algorithm is that each backend performs a partial join with its portion of source records and all target records. Then, the results are sent to the controller. The controller will then forward the final result to the host computer.

An area for further refinement concerns the designation of source and target relations for the join function. For our analysis, we assume that the number of the source and target records are equal. If this assumption changes, then the communication complexity and the access complexity of our proposals will be affected. Clearly, transmitting the small number records decreases the communication complexity. The effect on access complexity is less clear. The access complexity for the straightforward join algorithm is sensitive to the size of the file resident in main memory. Therefore, it may be desirable to select the larger of the two files as the file to be transmitted.

This thesis provides the groundwork for further analysis. We have presented computing, access, and communication complexities separately. If some relative weights can be assigned to these complexities, further analyses to evaluate the tradeoffs may lead to providing a choice among several alternatives, depending on the distribution of the relevant records among the backends, the communication cost and the acess complexity.

LIST OF REFERENCES

1. Kerr S. Douglas, Orooji Ali, Zong-Zhi S., Strawser R. Paula, The Implementation of a Multi-Backend Database System (MDBS): Part I - SOFTWARE ENGINEERING STRATEGIES AND EFFORTS TOWARDS A PROTOTYPE MDBS, Technical Report N00014-75-C-0573, Naval Postgraduate School, June 1983.

2. He X., Hsiao D. K., Kerr S. Douglas, Strawser R. Paula, The Implementation of a Multi-Backend Database System (MDBS): Part II - FIRST PROTOTYPE MDBS AND THE SOFTWARE ENGINEERING EXPERIENCE , Tech. Rep. N00014-75-C-0573, Naval Postgraduate School, July 1982.

3. Boyne, R. D., Hsiao D. K., Kerr S. Douglas, Steven A. D., The Implementation of a Multi-Backend Database System (MDBS): Part III - THE MESSAGE-ORIENTED VERSION WITH CONCURRENCY CONTROL AND SECONDARY-MEMORY-BASED DRECTORY MANAGEMENT , Tech. Rep. N00014-75-C-0573, Naval Postgraduate School, March 1983.

4. Merrett T. H., Why Sort-Merge Gives the Best Implementation of the Natural Join., McGill University, Montreal, Tech. Rep. SOCS-81-37, October 1981.

5. Dina B., David J. DeWitt, "Duplicate Record Elimination in Large Data Files", ACM Transactions on Database Systems , Vol.8,No.2., pages 255-265, June 1983.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defence Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4. Curricula Officier, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943	1
5. Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
6. Dr. Paula Strawser, 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
7. Turk Deniz Kuvvetleri Egitim Daire Baskanligi Bakanliklar Ankara TURKEY	3
8. Ltjg. Serdar Muldur Yildirim Mahallesi Civiciler Sokak Sandikcioglu Apt. No.2 Balikesir TURKEY	2
9. Istanbul Bogazici Universitesi Bilgisayar Bolumu Istanbul TURKEY	1